Resource-Predictable and Efficient Monitoring of Events

Jonas Mellin

Computer Science
School of Humanities and Informatics
University of Skövde
P.O. Box 408
SE-541 28 Skövde
Sweden

jonas.mellin@ida.his.se

June 2004
To Johanna, Agnes, and Clara
Carpe Diem!
Abstract

We present a formally specified event specification language (Solicitor). Solicitor is suitable for real-time systems, since it results in resource-predictable and efficient event monitors. In event monitoring, event expressions defined in an event specification language control the monitoring by matching incoming streams of event occurrences against the event expressions. When an event expression has a complete set of matching event occurrences, the event type that this expression defines has occurred. Each event expression is specified by combining contributing event types with event operators such as sequence, conjunction, disjunction; contributing event types may be primitive, representing happenings of interest in a system, or composite, specified by event expressions.

The formal specification of Solicitor is based on a formal schema that separates two important aspects of an event expression; these aspects are event operators and event contexts. The event operators aspect addresses the relative constraints between contributing event occurrences, whereas the event contexts aspect addresses the selection of event occurrences from an event stream with respect to event occurrences that are used or invalidated during event monitoring. The formal schema also contains an abstract model of event monitoring. Given this formal specification, we present realization issues of, a time complexity study of, as well as a proof of limited resource requirements of event monitoring.

We propose an architecture for resource-predictable and efficient event monitoring. In particular, this architecture meets the requirements of real-time systems by defining how event monitoring and tasks are associated. A declarative way of specifying this association is proposed within our architecture. Moreover, an efficient memory management scheme for event composition is presented. This scheme meets the requirements of event monitoring in distributed systems. This architecture has been validated by implementing an executable component prototype that is part of the DeeDS prototype.

The results of the time complexity study are validated by experiments. Our experiments corroborate the theory in terms of complexity classes of event composition in different event contexts. However, the experimental platform is not representative of operational real-time systems and, thus, the constants derived from our experiments cannot be used for such systems.

keywords: composition, efficiency, event, formalization, monitoring, performance, predictability, real-time systems, time complexity, timeliness
Acknowledgements

Ibi Semper Est Victoria Ubi Concordia Est (Eng., Where there is collabora-
ration there is always victory).

First, I would like to thank my thesis advisor Prof. Sten F. Andler who
has guided me through many invigorating and, sometimes, heated debates.
Then, Dr. Anthony Galton has provided invaluable input concerning for-
malisms that enabled me to complete this work. Prof. Sharma Chakravarthy
(co-advisor) has provided me with information, numerous hints as well as
source code, which have given me a great opportunity to experiment with
the topic. I thank Dr. Ulf Nilsson (co-advisor) for his valuable input con-
cerning formalisms as well as being supportive and caring. Additionally, I
thank Prof. Peter Fritzon (co-advisor) for his early input to this work.

Apart from the aforementioned people, several people have reviewed bits
and pieces of this Ph.D. thesis (in alphabetical order): Mikael Berndtsson,
Marcus Brohede, Jan Carlson, Bengt Eftring, Henrik Engström, Ammie
In particular, Birgitta Lindström, whom has given insightful critique regard-
ing the proof outlines as well as other formal issues, and Robert Nilsson, who
has provided valuable input concerning the content and presentation. Mikael
Berndtsson has shown an interest in event monitoring for which I am grate-
ful. Henrik Engström has reviewed articles that are the foundation of this
Ph.D. thesis. Ruth Morrison-Svensson did the language review, from whom
I learnt new things about the English language.

Moreover, I appreciate Dr. Brian Lings for the calm atmosphere he brings
with him everywhere he goes and Dr. Ajit Narayanan who introduced me
to artificial intelligence, philosophy, and Gottlobb Frege.

The DRTS group has been valuable to me, both its current members:
Marcus Brohede, Ammie Eriksson, Henrik Grimm, Mats Grindahl, Sanny
Gustavsson, Birgitta Lindström, Robert Nilsson, as well as old members
such as Ragnar Birgisson and Egir Leifsson. In particular, I thank Joakim
Eriksson for his good taste in music, delightful performances on the guitar,
and for being an excellent programmer; and I give my gratitude to Jörgen
Hansson for late night discussions, night club visits (in particular, the last
one), and the nail I received at your defence.

One of my firmest and earliest supporters is Stig Emanuelsson, head of
the former department of computer science at University of Skövde between
1986–2002. Without his support neither this Ph.D. thesis would have been
initiated nor would I have been a member of the computer science depart-
ment during its early heydays. Further, Lars-Erik Johansson, the principal of University of Skövde between 1977-2002, has supported my work.

Also, I would like to thank Anders Malmsjö for his delightful viewpoints concerning systems theory. Moreover, the gang at the department of computer science, University of Skövde has provided ample opportunity to do more or less sane things. Therefore, I give my gratitude to Mikael Berndtsson (again), Mikael Bodén, Lennart Börjesson, Bengt Eftring, Mikael Johannesson, Ingi Jonasson, Patrik Johansson, Björn Lundell, Dan Lundh, Lars Niklasson, Björn Olsson, and Anne Persson. I would also like to thank Lillemor Wallgren and Britt-Inger Karlsson for their administrative support.

Further, my family has withstood an enormous pressure stemming from this work and I thank them for their love, patience, support as well as kicking and dragging me off on vacations when I needed them.

To my friends, in particular Mats Boberg, Poul Möller, Pär Gärden, thanks for being there although this work impaired my social life.

This work has been supported by The Swedish Agency for Innovation Systems (VINNOVA, formerly NUTEK), The Ministry of Education and Science, The Swedish Foundation for Strategic Research (SSF, through ARTES and FLEXCON), and University of Skövde.

\LaTeX{} has been used to produce this Ph.D. thesis.

List of Publications

This Ph.D. thesis is based on these publications and Section 1.5 on p. 6 describes how these publications are included in this Ph.D. thesis.


# Table of Contents

1 Introduction ........................................ 1  
  1.1 Formalization of Event Monitoring .............. 3  
  1.2 Architectural Issues ............................ 5  
  1.3 Empirical Study ................................... 6  
  1.4 Brief Overview of Contributions ................. 6  
  1.5 Guide to the Mysteries ............................ 6  

I Issues in Timely Monitoring of Events ............... 9  

2 Event Monitoring Overview ............................ 11  
  2.1 Structure of Event Monitoring ................. 12  
    2.1.1 Event Detection and Composition in Event Monitoring 14  
    2.1.2 Event Subscription .......................... 15  
    2.1.3 Probe Effect and Observability .............. 15  
    2.1.4 Synchronous and Asynchronous Monitoring ......... 16  
    2.1.5 Hardware, Software, and Hybrid Event Monitors ... 17  
    2.1.6 Resource Requirements of Event Monitoring ....... 17  
    2.1.7 Optimizations of Event Monitoring ............ 18  
  2.2 Management of Event Monitoring ................. 19  
    2.2.1 Instrumentation of a Monitored Object ......... 20  
  2.3 The Notion of Time in Analysis Event Logs ....... 22  

3 Event Monitoring in Distributed and Real-Time Systems 25  
  3.1 Issues in Real-Time Systems ...................... 26  
    3.1.1 Timeliness ................................. 27  
    3.1.2 Event Shower Management .................... 28  
    3.1.3 Declarative Monitoring in Real-Time Systems .... 29  
    3.1.4 Probe Effect in Real-Time Systems .......... 30  

iv
## Table of Contents

3.2 Issues in Distributed Systems .................................. 31  
3.2.1 Distributed Real-Time Systems ............................... 31  
3.2.2 Global Time Base and Global Ordering ..................... 32  
3.3 Event Monitoring for Fault Management ........................ 32  
3.4 Active Real-Time Databases .................................... 33  
3.4.1 Transactions and Coupling Modes ............................ 34  
3.5 Patterns of Events and Transactions ............................ 35  
3.6 Event Monitoring for Real-Time Systems ...................... 36  

4 Event Composition: Syntax, Semantics, and Processing .......... 39  
4.1 Addressed Prototypes and Event Specification Languages ....... 41  
4.2 Basic Formal Definitions ....................................... 42  
4.3 Syntax .......................................................... 42  
4.3.1 Abstract Event Expression Syntax ........................... 43  
4.3.2 Evaluation of the Abstract Event Expression Syntax ....... 44  
4.3.3 Examples of Primitive Event Types .......................... 45  
4.4 Semantics of Event Operators ................................... 46  
4.4.1 Interval and Termination Semantics ......................... 50  
4.5 Event Contexts: Order, Reuse, and Time Constraints .......... 51  
4.5.1 Basic Formal Definitions for Event Contexts ............... 52  
4.5.2 Initiator and Terminator Occurrences of Event Types ..... 54  
4.5.3 General Event Context ..................................... 55  
4.5.4 The Semantics of Event Contexts ............................ 55  
4.5.5 Overlapping and Non-overlapping Contexts ................. 61  
4.5.6 Time Constraints ........................................... 62  
4.5.7 Combining Constraints ...................................... 63  
4.6 Managing Concurrent Event Occurrences ....................... 63  
4.7 Formalization of Event Composition: Existing Approaches .... 65  
4.8 Evaluation of Event Specification Languages .................. 66  
4.8.1 Support for Event Contexts and Time Constraints ......... 66  
4.8.2 Evaluation of Existing Event Operators .................... 67  
4.9 Architectural Issues of Event Detection ....................... 75  
4.9.1 Mapping Event Types to Tasks .............................. 77  
4.9.2 Invocation, Scheduling, and Delivery ....................... 77  
4.9.3 Event Composition Mechanism and Event Subscription ..... 78  
4.9.4 Memory Management ........................................ 82  
4.9.5 Event Parameters ........................................... 83  
4.9.6 Timeout Service ............................................. 84
## 4.10 Event Criticality ........................................ 85

## 5 Problem Definition ........................................... 87
  5.1 Purpose .................................................. 87
  5.2 Motivation ............................................... 88
  5.3 Assumptions and their Implications ...................... 88
  5.4 Strategy and Goals ....................................... 90

### II Semantics of Event Composition ................................. 93

### 6 Formal Specification of Event Composition ......................... 97
  6.1 The Basis for an Event Composition Schema ................. 97
  6.1.1 The Problem of Event Contexts ......................... 98
  6.2 The Event Composition Schema ................................ 102
  6.2.1 Basis for Constructing Event Operator Rules .......... 112
  6.2.2 Event Expression Property Functions .................. 113
  6.2.3 Constructing Context Predicates ....................... 115
  6.2.4 Constructing Operator Rules ............................ 116
  6.3 The Solicitor Event Specification Language .................. 120
  6.3.1 Basic Assumptions Concerning Event Types .............. 120
  6.3.2 Expiration Semantics .................................. 124
  6.3.3 Event Expression Property Functions in Solicitor .... 124
  6.3.4 Context Predicates in Solicitor ........................ 126
  6.3.5 Operator Rules ....................................... 134
  6.4 Discussion ............................................... 139
  6.4.1 Alternative Way to Expiration Semantics .............. 141
  6.4.2 Algebraic Properties of Event Operators .............. 141
  6.5 Related Work on Formalization of Event Composition ....... 142
  6.5.1 Formalization of Snoop over Event Histories .......... 143
  6.5.2 Formalization of Snoop using First Order Predicate Logic ................................................. 145
  6.5.3 Interval-Based Algebra for Restricted Event Detection 146
  6.5.4 ODE and SMILE ........................................ 147
  6.5.5 Monitoring of Real-Time Logic .......................... 147
  6.5.6 Active Database Formalisms ............................. 148
  6.5.7 Event Composition Formalism Based on Deductive Databases .................................................. 148
## TABLE OF CONTENTS

6.5.8 Decorating Event Specification Languages .................................. 148
6.6 Summary .......................................................................................... 149

7 Realization Issues ............................................................................. 151
  7.1 Connector Set Abstraction for Event Composition ......................... 152
  7.1.1 Invariant for Implicit Invocation ............................................. 154
  7.1.2 Example of Usage of Invariant ................................................ 165
  7.1.3 Invariant for Explicit Invocation ............................................. 166
  7.2 Scheduling of Evaluation of Event Composition ........................... 167
  7.2.1 Data-driven Scheduling .......................................................... 168
  7.2.2 Control-Driven Scheduling of Event Composition ................... 171
  7.3 Discussion ...................................................................................... 174
  7.4 Related Work on Realization ........................................................ 174
  7.5 Summary ....................................................................................... 175

8 Bounded Event Composition ............................................................. 177
  8.1 Prerequisites of Early Detection of Expiration ............................... 180
    8.1.1 Chronicle Aperiodic Operator ............................................... 180
    8.1.2 Aperiodic versus Chronicle Aperiodic Operator .................... 183
    8.1.3 Initialization Transformation ............................................... 184
  8.2 Proof of Early Detection of Expiration ........................................... 185
  8.3 Expiration Management ............................................................... 190
  8.4 Results ......................................................................................... 192
    8.4.1 Discussion ............................................................................. 192
  8.5 Related Work on Bounded Resources .......................................... 193
  8.6 Summary ....................................................................................... 194

9 Algorithmic Time Complexity of Event Monitoring .......................... 195
  9.1 Assumptions about Processing ..................................................... 197
    9.1.1 Implementation Assumptions ............................................... 200
  9.2 Sources of Complexity .................................................................. 201
    9.2.1 Motivation of Complexity Sources and their Model Coverage .................................................................................. 201
  9.3 Time Complexity Basics ............................................................... 203
    9.3.1 Semantics of Notation of Multiple Parameters ...................... 204
  9.4 Algorithmic Time Complexity of Event Monitoring ..................... 204
    9.4.1 Composition Cost Function .................................................. 205
    9.4.2 Prepare for Delivery Cost Function ....................................... 209
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3.3 Algorithmic Time Complexity for Event Contexts</td>
<td>209</td>
</tr>
<tr>
<td>9.4.4 Recent Event Context</td>
<td>213</td>
</tr>
<tr>
<td>9.4.5 Continuous Event Context</td>
<td>213</td>
</tr>
<tr>
<td>9.4.6 General Event Context</td>
<td>213</td>
</tr>
<tr>
<td>9.5 Discussion</td>
<td>214</td>
</tr>
<tr>
<td>9.5.1 Relevance of Study</td>
<td>214</td>
</tr>
<tr>
<td>9.5.2 Remaining Issues</td>
<td>215</td>
</tr>
<tr>
<td>9.6 Related Work on Algorithmic Complexity</td>
<td>215</td>
</tr>
<tr>
<td>9.6.1 Chronicle Recognition</td>
<td>215</td>
</tr>
<tr>
<td>9.6.2 Monitoring of Real-time Logic</td>
<td>216</td>
</tr>
<tr>
<td>9.7 Summary</td>
<td>217</td>
</tr>
<tr>
<td>III Architectural Issues in Event Monitoring</td>
<td>219</td>
</tr>
<tr>
<td>10 Architecture of Resource-Predictable Event Monitoring</td>
<td>223</td>
</tr>
<tr>
<td>10.1 Overview of the Architecture</td>
<td>223</td>
</tr>
<tr>
<td>10.1.1 Main Scenarios</td>
<td>226</td>
</tr>
<tr>
<td>10.1.2 Main Configurations</td>
<td>226</td>
</tr>
<tr>
<td>10.1.3 Event Monitor Specification</td>
<td>227</td>
</tr>
<tr>
<td>10.2 Design Choices</td>
<td>230</td>
</tr>
<tr>
<td>10.3 Maintenance Issues</td>
<td>230</td>
</tr>
<tr>
<td>10.3.1 Management of Event Monitoring</td>
<td>230</td>
</tr>
<tr>
<td>10.3.2 Fault Tolerance</td>
<td>231</td>
</tr>
<tr>
<td>10.4 Issues Concerning Event Types, Tasks, and Transactions</td>
<td>232</td>
</tr>
<tr>
<td>10.4.1 Association of Event Types and Event Monitoring Tasks</td>
<td>232</td>
</tr>
<tr>
<td>10.4.2 Association of Tasks and Transactions</td>
<td>232</td>
</tr>
<tr>
<td>10.4.3 Isolation Property of Transactions and Event Monitoring</td>
<td>233</td>
</tr>
<tr>
<td>10.5 Event Subscription</td>
<td>233</td>
</tr>
<tr>
<td>10.6 Event Pruning</td>
<td>234</td>
</tr>
<tr>
<td>10.6.1 Management of Concurrent Event Occurrences</td>
<td>234</td>
</tr>
<tr>
<td>10.6.2 Aggregation of Event Occurrences Based on Timeouts</td>
<td>235</td>
</tr>
<tr>
<td>10.6.3 Aggregation of Duplicates</td>
<td>235</td>
</tr>
<tr>
<td>10.7 Event Operators, Algorithms, and Mechanisms</td>
<td>235</td>
</tr>
<tr>
<td>10.8 Timeout Service</td>
<td>236</td>
</tr>
<tr>
<td>10.9 Event Parameter Management</td>
<td>236</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

10.10 Optimizations .................................................. 236
10.11 Discussion ................................................... 237
10.12 Related Work .................................................. 238
   10.12.1 Snoop .................................................. 238
   10.12.2 HiPAC, REACH, SAMOS, ODE, NAOS .............. 239
   10.12.3 Schwidersky’s Monitoring of Distributed Systems .. 240
   10.12.4 SMILE .................................................. 240
   10.12.5 Event Pattern Language .................................. 241
   10.12.6 GEM, Hermes, COBEA, X^2TS ......................... 241
   10.12.7 Monitoring using Real-Time Logic ..................... 243
   10.12.8 Chronicle Recognition .................................. 244
   10.12.9 Temporal Calculi for Communication Systems ....... 244
10.13 Summary ..................................................... 244

11 Memory Management of Event Composition ................. 245
   11.1 Memory Management Problem ............................... 246
   11.2 Self-Referencing Memory with Safe Pointers ........... 248
      11.2.1 Abstract Data Types ................................. 248
      11.2.2 Self-Referencing Memory ............................ 250
      11.2.3 Safe Pointers ........................................ 250
      11.2.4 Optimizing For Cloning .............................. 251
      11.2.5 Lazy Composition of User Parameters ............... 251
      11.2.6 Aggregation for Correct Signalling ................ 251
   11.3 Realization in C++ ......................................... 252
   11.4 Empirical Study ........................................... 252
   11.5 Applicability .............................................. 253
   11.6 Related Work ............................................... 254
   11.7 Future Work ............................................... 254

IV Results ......................................................... 257

12 Empirical Studies ............................................... 261
   12.1 Foundation of the Experiments ............................ 261
      12.1.1 Experimental Target Platform ........................ 262
      12.1.2 Overall Design of Experiment ......................... 262
      12.1.3 Variables .............................................. 264
      12.1.4 Managing Factors of Experimental Resources ....... 265
TABLE OF CONTENTS

12.1.5 Enabled Analysis of Collected Data ............... 267
12.2 Validation of Algorithmic Time Complexity .......... 267
  12.2.1 Distribution of Response Times ................. 268
  12.2.2 Dependency on Event Type Size ................. 268
  12.2.3 Dependency of the Size of the Filtered Event Log 270
  12.2.4 Dependency on Parameter Size ................. 272
12.3 Discussion ....................................... 274
  12.3.1 Relevance of Parameter Choices ................. 274
  12.3.2 Accuracy of Results .......................... 276
  12.3.3 Comparison to Other Experiments in this Work .... 276
12.4 Summary ........................................... 276

13 Discussion ........................................ 277
  13.1 Evaluation of Event Composition in Real-Time Systems 277
    13.1.1 Suitability: Achieving Timeliness ............ 278
    13.1.2 Usefulness: Alternative Methods .............. 281
  13.2 Formalization of Event Monitoring ................. 284
    13.2.1 Alternative Strategies to Formalization .... 285
    13.2.2 Forward and Backward Deduction .............. 286
  13.3 Order and Time .................................. 287
  13.4 Performance and Efficiency Considerations .......... 288
  13.5 Support for Real-Time Systems ..................... 290
  13.6 Application Examples ................................ 291
    13.6.1 Fault Management ............................ 291
    13.6.2 Active Databases ............................ 293

14 Related Work ..................................... 297
  14.1 Formalization of Event Composition ................. 297
  14.2 Realization Issues from a Formal Specification .... 300
  14.3 Bounded Resource Requirements of Event Composition 301
  14.4 Algorithmic Time Complexity of Event Composition 302
  14.5 Architecture of Event Monitoring .................. 303

15 Conclusions ...................................... 305
  15.1 Summary .......................................... 305
  15.2 Contributions .................................... 307
    15.2.1 Formalized Schema for Event Composition ..... 309
    15.2.2 Proof of Concept: Solicitor ................ 309
TABLE OF CONTENTS

15.2.3 Bounded Event Composition . . . . . . . . . . . . . . 310
15.2.4 Algorithmic Time Complexity of Event Composition 311
15.2.5 Architectural Issues Concerning Event Monitoring . . 311
15.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . 312
15.3.1 Comparing Event Specification Languages . . . . . . 312
15.3.2 Automatic Translation to other Models . . . . . . . . 312
15.3.3 Generic Event Specification Language . . . . . . . . . 313
15.3.4 Adopt Time Constraint Preprocessing in Solicitor . . 313
15.3.5 Hybrid Mechanism for Event Composition . . . . . . 313
15.3.6 Pruning of Event Occurrences . . . . . . . . . . . . . 313
15.3.7 Execution Time Analysis . . . . . . . . . . . . . . . . . 314
15.3.8 Architecture for Maintainable Efficient Event
Composition . . . . . . . . . . . . . . . . . . . . . . . . 315
15.3.9 Recovery and Event Composition . . . . . . . . . . . 316
15.3.10 Automatic Generation of Event Monitor
Specifications . . . . . . . . . . . . . . . . . . . . . . . 316
15.3.11 Providing Observability for Testing . . . . . . . . . . 317
15.3.12 Relaxing the Requirements of Event Monitoring . . 317
15.3.13 Parameterized Event Types . . . . . . . . . . . . . . . 317

V Bibliography and Appendices 319

References 321

A Formal Notation Convention and Symbol Dictionary 335
A.1 Formal Notation Convention . . . . . . . . . . . . . . . . . 335
A.2 Symbol Dictionary . . . . . . . . . . . . . . . . . . . . . . . . 336

B Solicitor Semantics 345
B.1 Solicitor Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . 345
B.1.1 The Complete Event Expression Syntax . . . . . . . . 345
B.1.2 The Event Monitor Specification Syntax . . . . . . . . 346
B.2 Basic Definitions . . . . . . . . . . . . . . . . . . . . . . . . . 347
B.3 Solicitor Basics . . . . . . . . . . . . . . . . . . . . . . . . . . 355
B.4 Solicitor Event Property Functions . . . . . . . . . . . . . 357
B.5 Event Context Predicates . . . . . . . . . . . . . . . . . . . . 359
B.6 Event Operator Rules . . . . . . . . . . . . . . . . . . . . . . . 363
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Programming Logic</td>
<td>369</td>
</tr>
<tr>
<td>D Distribution of Response Times</td>
<td>371</td>
</tr>
<tr>
<td>E Recent and Chronicle Dependent on Size of Type</td>
<td>383</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Overview of event monitoring structure .......................... 13
2.2 Event monitoring process [Mel00] ................................. 14
2.3 Time: present, past, and future ................................. 22
2.4 Example of right-branching time ................................. 24
2.5 Example of left-branching time ................................. 24
3.1 A view of (distributed) real-time systems [KV94] ............ 27
4.1 Result of composition of $E_{12} = E_1; E_2$ in general event context 56
4.2 Composition of $E_{12} = E_1; E_2$ in recent event context .... 58
4.3 Composition of $E_{12} = E_1; E_2$ in chronicle event context ... 58
4.4 Composition of $E_{12} = E_1; E_2$ in continuous event context ... 59
4.5 Composition of $E_{12} = E_1; E_2$ in cumulative event context ... 60
4.6 Event graph representation of $(E_1; E_2)E_3$ .................... 79
4.7 Procedural definition example in Snoop [Ada02] ............. 81
4.8 Petri-net representation of $(E_1; E_2)E_3$ [Gat94, pp. 101-102] ... 82
4.9 Finite state automata representation of $(E_1; E_2)E_3$ ........ 82
4.10 Example of directed acyclic graph for a Chronicle specification 83
5.1 Assumed event monitoring structure ............................. 89
6.1 Sequence operator algorithm for chronicle event context ... 99
6.2 Basic production system algorithm [Nil82, p. 21] ............. 101
6.3 Depiction of Example 6.1 on p. 103 in general, recent, and chronicle event context ......................... 103
6.4 Depiction of recent filter in Snoop ............................ 144
7.1 Proof outline of the sequence operator algorithm ............ 176
8.1 Sequence operator algorithm for chronicle event context with timeout management .......................... 179
9.1 Simplified realization of the least constrained event composition algorithm ........................................... 197
9.2 Algorithm performing composition only ................................. 199
10.1 Task perspective of the event monitoring architecture ........ 224
10.2 Client layers of the event monitoring architecture ............. 224
10.3 Server layers of the event monitoring architecture ............ 225
10.4 Event monitor specification example ................................. 229
11.1 Directed acyclic graph, representing an event occurrence of $E_1; E_2$ [Mel00] ....................................... 246
11.2 Abstract data types for event occurrences (based on [Mel00]) 249
12.1 Response time dependent on type size for general event context 269
12.2 Response time dependency on history size in chronicle context 271
12.3 Response time dependency on history size in recent context . 272
12.4 Response time dependency on history size in general context 273
12.5 Response time dependency on parameter size with unsafe pointer in recent context ............................... 274
12.6 Response time dependency on parameter size with safe pointer in recent context ................................. 275
13.1 Suitability of event composition for process control ........... 279
13.2 Suitability of event composition for air traffic control and training simulation ................................. 280
13.3 Finite state automaton for a sequence of $E_1$ followed by $E_2$ 283
13.4 Alternative route example [Mel98a] ................................. 295
15.1 Parameterized event monitor specification example ............. 318
D.1 Distribution of response times for chronicle context, $|E|=1$, $|G|=10$ 372
D.2 Distribution of response times for chronicle context, $|E|=1$, $|G|=100$ 372
D.3 Distribution of response times for chronicle context, $|E|=1$, $|G|=1000$ 373
D.4 Distribution of response times for chronicle context, $|E|=1$, $|G|=10000$ 373
D.5 Distribution of response times for chronicle context, $|E|=10$, $|G|=10$ 374
D.6 Distribution of response times for chronicle context, $|E|=10$, $|G|=100$ 374
D.7 Distribution of response times for chronicle context, $|E|=10$, $|G|=1000$ 375
D.8 Distribution of response times for chronicle context, $|E|=10$, $|G|=10000$ 375
D.9 Distribution of response times for chronicle context, $|E|=50$, $|G|=10$ 376
D.10 Distribution of response times for chronicle context, $|E|=50$, $|G|=100$ 376
D.11 Distribution of response times for chronicle context, $|E|=50$, $|G|=1000$ 376
D.12 Distribution of response times for recent context, $|E|=1$, $|G|=10$ 377
D.13 Distribution of response times for recent context, $|E|=1$, $|G|=100$ 378
D.14 Distribution of response times for recent context, $|E|=1$, $|G|=1000$ 378
D.15 Distribution of response times for recent context, $|E|=1$, $|G|=10000$ 379
D.16 Distribution of response times for recent context, $|E|=10$, $|G|=10$ 379
D.17 Distribution of response times for recent context, $|E|=10$, $|G|=100$ 380
D.18 Distribution of response times for recent context, $|E|=10$, $|G|=1000$ 380
D.19 Distribution of response times for recent context, $|E|=10$, $|G|=1000$ 381
D.20 Distribution of response times for recent context, $|E|=50$, $|G|=10$ 381
D.21 Distribution of response times for recent context, $|E|=50$, $|G|=100$ 382
D.22 Distribution of response times for recent context, $|E|=50$, $|G|=1000$ 382
E.1 Response times for recent context, $|G|=1000$ 384
E.2 Response times for recent context, $|G|=5000$ 384
E.3 Response times for recent context, $|G|=10000$ 385
E.4 Response times for chronicle context, $|G|=1000$ 385
E.5 Response times for chronicle context, $|G|=5000$ 386
E.6 Response times for chronicle context, $|G|=10000$ 386
## List of Tables

2.1 Summary of monitor types ........................................ 18

4.1 Existing attributes in Solicitor .................................. 44

4.2 Summary of evaluation of abstract syntax for active database prototypes .................................................. 45

4.3 Summary of evaluation of abstract syntax for formal approaches ...................................................... 46

4.4 Difference between event contexts ................................. 60

4.5 Applicable event contexts in available formalizations ....... 65

4.6 The support of event specification languages for event contexts and time constraints .............................. 67

4.7 The support of event specification languages for overlapping event contexts without time constraints ........... 68

4.8 Operator mechanism for event specification languages .... 79

4.9 Existing composition mechanisms in the event specification languages ..................................................... 80

6.1 Proof table for the sequence operator ......................... 136

6.2 Event type sets for sequence, conjunction, and disjunction event operators .................................................. 138

6.3 Proof table for the conjunction operator ....................... 139

6.4 Proof table for the disjunction operator ....................... 140

7.1 Restriction category of initiators and terminator occurrences 165

8.1 Proof table for the chronicle aperiodic operator .......... 183

9.1 Variables for Time Complexity Investigation .................. 201

9.2 Time complexity of event composition in chronicle event context ......................................................... 211

9.3 Complexity of continuous event context ........................ 213
LIST OF TABLES

10.1 Summary of Solicitor features ........................................ 239
11.1 Event history example ..................................................... 246
12.1 History variation for different $|E|$ in general event context .... 270
13.1 Time constraints of updates based on Locke’s work [Loc96] .... 278
A.1 Dictionary of Mathematical Symbols ................................. 336
B.1 Proof table for the non-occurrence operator ...................... 366
B.2 Proof table for the aperiodic operator .............................. 367
Chapter 1

Introduction

“Omnia mutantur, nihil interit.”

/Neil Gaimann

A reactive system reacts to stimuli in an environment (e.g., controlled process) and needs to respond (e.g., control something in the environment) to these stimuli. A real-time system is a reactive system that need to respond to stimuli in a finite period of time. In other words, real-time systems must be timely. Since these stimuli can be viewed as events (or event occurrences), event monitoring is desirable in real-time systems. Examples of real-time systems are aircraft mission control systems, car (mission) control systems, spacecraft mission control systems, production control systems, telephone switching systems, and chemical process systems. Many of these systems are critical and a missed deadline may lead to catastrophic results, such as people dying or great economic loss.

Event monitoring provides means for collecting, analyzing, and signalling event occurrences to subscribers. In a real-time system, these subscribers are tasks executing algorithms that control the environment based on the (event) stimuli. For example, an automatic door should open within $d$ duration after the real-time system has detected that the event “someone approaches the door” has occurred. Another example is when a driver in a car indicates that full acceleration is needed, then the automatic gear box should shift down within $d'$ duration if conditions so admit. In both these cases, we need a
sensor detecting the occurrence of the event, an event monitor that collects these event occurrences, analyses them, and passes on the analyzed event occurrences to the subscribing tasks. These tasks should then perform an appropriate action in response to the event occurrences.

Reactive systems may be prone to event showers [KV94], that is, a burst of events, often caused by a single physical event, that may cause an overload in the system. An overload in a real-time system implies that the resources used by the tasks are inadequate to guarantee that these tasks can meet their time constraints. This situation is commonplace in, for example, train traffic control systems where several sensors may generate redundant and repeated events at the same time that threaten to congest the system. Event monitoring is necessary in reactive systems that are prone to event showers, since it can be used to aggregate, compact and discard event occurrences before they are signalled to tasks. Effectively, this can reduce the resource load caused by the tasks.

Reactive systems can be constructed to avoid event showers. However, this only works for environments where we have extensive knowledge about the controlled process. For example, a chemical process where the temperature should be at a certain level is such an environment. In contrast, a train traffic control system is an environment where it is impossible to have sufficient knowledge to avoid event showers, since redundant sensors allow redundant and repeated events to be signalled to the train traffic control system effectively impairing its ability to handle all events. Avoidance require adequate resources for the worst-case situation. Even if we have sufficient knowledge about the worst-case situation in an environment, the difference between the average-case and worst-case situation may be so large that an unacceptable portion of the capacity of resources is wasted during the operation of the real-time system if event showers are to be avoided.

To use event monitoring for real-time systems, we need to know that timeliness is not violated. To ensure timeliness, the major methods in real-time systems engineering are to split an application into tasks with predictable resource requirements and schedule these tasks based on their time constraints on resources allocated to these tasks. Since event monitoring is performed by one or more tasks that must have predictable resource requirements, it is necessary to know that there is a bound on the number of steps an algorithm performing event monitoring must perform. Moreover, the total task load including the monitoring task load must be schedulable in the sense that all time constraints can be met. To ensure this, all
1.1 Formalization of Event Monitoring

algorithms executed by tasks must also be sufficiently efficient. In other words, no algorithm should cause unacceptable overhead for the required functionality. Further, the algorithms should preferably scale well in terms of response time with respect to the size of the input (e.g., the number and type of event occurrences).

To demonstrate predictable resource requirements of event monitoring, it is necessary to define the process formally to provide a proof. Moreover, to address efficiency, the asymptotic efficiency (also known as scalability) of the algorithms with respect to response time needs to be studied. The reason why efficiency in itself is not studied is that it requires knowledge about application requirements as well as specific constraints on the technical solution such as processor performance, memory performance etc. In other words, it is impossible to achieve generic results for efficiency.

1.1 Formalization of Event Monitoring

The analysis in this Ph.D. thesis of event monitoring is based on a declarative event specification language such as:

- TQuel (Temporal Query Language) [Sno88]
- Snoop (e.g., [CM94, CKAK94])
- COMPOSE [GJS93]
- SAMOS (Swiss Active Mechanisms Based Object-Oriented Database System) (e.g., [GD92, GD93, Gat94])
- TCCS (Temporal Calculi for Communicating Systems) [BG97]
- Monitoring of Real-Time Logic (RTL) (e.g., [ML97a, ML97b, LMK98])
- Chronicle recognition (e.g., [DGG93, Dou96, Gha96])

There are several advantages of using a declarative approach: instrumentation of the monitored object can be automatic [Buc94]; it is possible to analyze the event specification in terms of correctness (e.g., guaranteeing that the specification contains no invalid time constraints [ML97a, ML97b]), algorithmic time complexity (e.g., [ML97a, ML97b, DGG93]) etc.; less effort is required to introduce changes [Sno88]. A declarative event specification
language consists of event operators such as sequence (that requires two event occurrences to be in a sequence) combining primitive (i.e., pre-defined happenings of interest) or composite (i.e., defined in terms of an event expression) events into patterns as well as event contexts controlling the selection of contributing event occurrences with respect to used and invalidated event occurrences. For example, an event context can be to always select the most recent event occurrences of contributing event types of a composite event type. The reason for event contexts is to improve response times of event monitoring. In contrast, in imperative monitoring [Sno88], instrumentation, analysis of the data generated by instrumentation, and dissemination of results of the analysis has to be implemented manually using an imperative programming language.

The major problem in formalization of event monitoring is to define the computation mathematically in such a way that the formalism is parameterized on event contexts. Several event specification languages such as RTL, TCCS, Chronicle recognition specifications, are based on theories limited to specific event contexts. Therefore, proofs performed within these theories are limited to those same event contexts. In contrast, event specification languages such as Snoop and its derivatives support several event contexts, but none of these languages has been defined in such a way that processing can be analyzed to prove, for example, that there is a bound on the number of steps the analysis must perform.

To enable analysis of event monitoring in terms of predictability and efficiency, a formalized schema where each event operator is defined by a set of rules, is introduced. These rules are parameterized on event-context predicates describing the effects of event contexts. The schema defines necessary requirements on these rules to produce correct results.

Solicitor [MA02, Mel98a], an event specification language based on Snoop (e.g., [CM94, CKAK94]), has been specified within this schema. It is a demonstration of how the schema can be used. Vital invariants as well as correct scheduling of different parts of the analysis of event occurrences have been derived from the specification of Solicitor. Moreover, in Solicitor, it is possible to prove that the event monitoring based on any combination of event operators and event contexts requires a limited number of steps to complete, given that there is a minimum interarrival time between event occurrences as well as an expiration time associated with each event specification.
A model for time complexity of event monitoring has been derived from the specification of Solicitor. This model of event monitoring is general in the sense that it covers other event specification languages. The accuracy of this model has been validated by performing experiments using an implementation of Solicitor. This implementation of Solicitor meets necessary requirements of real-time systems as well as satisfies invariants derived from the Solicitor specification.

1.2 Architectural Issues

The architecture of an event monitor for real-time systems must meet the requirements of the infrastructure (mainly, the operating system and runtime execution environment). To meet these requirements, event monitoring needs to be split into one or more tasks with assigned time constraints. Moreover, real-time systems are often embedded architectures that lack disks, keyboards, and displays. Therefore, it is desirable to compile application specification, operating system configuration etc. into object code such that these specifications and configurations are executed during initialization of the real-time systems rather than read from, for example, a file.

The Solicitor prototype is based on a client/server solution, where the client is based on procedure calls and the server part is a set of tasks performing the actual event monitoring. This separation is a requirement to allow the real-time system to schedule the tasks to meet the time constraints. The client interface is designed to minimize the overhead introduced into the monitored object such as an application, an operating system etc. The client interface is generated from an event monitor specification, in which monitoring of event specifications are grouped into monitoring tasks with different time constraints. By using the generated interface, many problems of generic interfaces necessary to handle any event type with the same functionality are avoided. Instead, each event type has specific functionality that is automatically generated. The event types are explicitly mapped to monitoring tasks. This gives greater control to real-time scheduling of tasks in terms of meeting deadlines.

The Solicitor prototype can be configured to do implicit or explicit invocation of event composition, control- or data-driven scheduling of evaluation of event types as well as delayed or immediate signalling of the results to event subscribers. In control-driven scheduling, the evaluation of event types
Introduction

are scheduled according to some order that guarantees correct results from the event monitor. In contrast, in data-driven scheduling, evaluation of event types are triggered when a result of contributing events is generated. In immediate signalling of the results, when a result has been computed, it is directly signalled to subscribers. In contrast, in delayed signalling, the results are not signalled to subscribers until all matching has been completed for the last invoked event composition.

1.3 Empirical Study

The accuracy of the theoretical model of time complexity is demonstrated for factors that are significant for real-time systems. The time complexity classes of different event contexts are corroborated by the experiments; moreover, the difference, in terms of complexity classes, between different event contexts is clearly visible. However, due to experimental errors in the performed experiments, it is not possible to obtain accurate constants (for the computational cost functions) that are representative for the target systems.

These experiments indicate that the size of the (filtered) event log and the size of event expression are the two most important parameters of the computational cost function of event monitoring. The use of safe pointers also give less overhead than unsafe pointers.

1.4 Brief Overview of Contributions

The contributions of this Ph.D. thesis is to formally define event composition processing to (i) prove that there is a bound on the processing, (ii) address algorithmic time complexity issues of event composition, (iii) address realization issues such as deriving necessary invariants for program verification, (iv) define an architecture for embedded real-time systems, (v) define an architecture for memory management for event monitoring, and (vi) validate the accuracy of the results from the theoretical study by experiments.

1.5 Guide to the Mysteries

In Part I, the major issues of predictable monitoring of event are presented. This presentation is arranged as follows: in Ch. 2, an overview of monitoring
and event composition in particular is described; in Ch. 3, requirements of real-time and distributed event monitoring as well as applications of monitoring in such systems are presented (this chapter is a revised version of the background in the licentiate thesis [Mel98a]); in Ch. 4, issues such as syntax, semantics, and pragmatics of event composition are discussed (where parts are revised from the licentiate thesis [Mel98a] and some parts are from our sections in the chapter by Mellin et al. [MEA01]); and finally, in Ch. 5, the problem of predictable monitoring of events is precisely defined and motivated.

The method and result are found in Part II, Part III, and Part IV. Each chapter in the first two parts contains solution and results with respect to the particular problem addressed in the chapter. In contrast, Part IV presents the results with respect to the whole problem and the whole solution.

Part II contains a definition of the formal semantics of event specification languages for predictable monitoring of events (in Ch. 6) that is a revised version of the work of Mellin and Andler [MA02], realization issues defining valid implementations (in Ch. 7), a proof of a bounded number of processing steps in event composition (in Ch. 8), as well as an investigation of the algorithmic time complexity of event composition (in Ch. 9). Part III defines the architecture of resource-predictable and efficient event monitoring for (embedded) real-time systems (in Ch. 10) (where initial ideas addressed by Mellin et al. [MEA01] are pursued); in particular, memory management is emphasized (in Ch. 11) based on the work of Mellin [Mel00] (with minor revisions).

Finally, Part IV contains an empirical study of a complete implementation of the event monitor architecture (in Ch. 12) and a discussion (in Ch. 13) addressing the whole problem and solution, for example, relevance of event composition for real-time systems. In Ch. 14, the whole problem, solution, and results are compared and contrasted to related work. Further, this part also includes the conclusions (in Ch. 15) that contains a summary, contributions, and future work.
Introduction
Part I

Issues in Timely Monitoring of Events
Chapter 2

Event Monitoring Overview

“What happens depends on our way of observing it or on the fact that we observe it.”

—Werner Heisenberg

The purpose of this chapter is to give an overview of event monitoring and, in particular, to pinpoint the general area of this Ph.D. thesis. This chapter also provides the basic ontology of event monitoring by introducing fundamental concepts.

The term monitor is used for various constructs with different meanings. In this Ph.D. thesis, an event monitor is a construct for the monitoring of events (or properties) of a system; this brief definition follows the definition of Snodgrass [Sno88]: monitoring is the extraction of dynamic information concerning the (computational) process. Two examples of event monitoring are the monitoring of violations of time constraints by Jahanian et al. [JRR94] and monitoring of events in real-time systems by Plattner [Pla84].

In this Ph.D. thesis, the term “event monitor” does not imply constructs whose semantics are of control rather than observation. An example of this latter meaning is found in concurrent programming\(^1\) where a monitor is an abstract data type providing the programmer with a way of synchronizing concurrent access to resources [Hoa74, And91]. Another example of the

\(^1\)According to Andrews [And91], concurrent programming is the fundament of distributed programming, parallel programming and multithreaded programming
latter meaning is that the term “monitor” has also been used as a synonym for operating systems by, for example, Silberschatz and Galvin [SG94].

2.1 Structure of Event Monitoring

Event monitoring can be viewed as if it is performed by a set of functions (sensing, data collection, and analysis). An event monitor is mainly a data collector that receives or fetches events (observations) from sensors (depicted in Fig. 2.1 on the facing page) and stores these observations in a database named filtered event log. This filtered event log is a subset of an event history without useless observations (e.g., observations that are too old to be meaningful for current or future decisions); an event history is defined to contain all observations. If an event monitor receives events from the sensor, then it is reactive. This is referred to as the push protocol. In contrast, if an event monitor fetches data from the sensor, then it is (pro)active. This is referred to as the pull protocol. The former is emphasized in this Ph.D. thesis, since the main emphasis is event monitoring for reactive systems. An event monitor can be realized as software, hardware or a hybrid between hardware and software [HW90].

A sensor is an instrumentation of a monitored object. For example, the instrumentation can be additional hardware that allows us to observe the signals on the processor bus [Pla84] or the instrumentation can be additional code that allows us to observe what happens in the monitored objects (e.g., ARTS operating system [TM89]). That is, the sensor registers some event of interest.

The sensor can sense these events directly or indirectly. For example, to monitor the state of the processor using the standard test interfaces (such as JTAG) or using an in-circuit (hardware) emulator of the processor gives us the ability for direct sensing, since it is possible to observe the internal events of a processor directly. In contrast, event monitoring of the processor state via a processor bus is indirect sensing, because the event monitor perceives the processor state indirectly via the signals on the bus\(^2\). A monitored object is a component that is desirable to observe, for example, an operating system, a processor, or a software component.

The observations are analyzed and presented to a subscriber such as a user or another component. The subscriber takes some action that is

\(^2\text{N.B. this is direct input/output sensing}\)
2.1 Structure of Event Monitoring

Monitored object

Sensors (sensing)

Receive (push)

Fetch (pull)

Analysis

Data collection

Write

Read/write

Monitor

Event log

Post mortem analyzer

a) Off-line

b) On-line

c) Hybrid

Figure 2.1: Overview of event monitoring structure

dependent on the result of the analysis. Fig. 2.1 represents two extremes of event monitoring. In (a), the event monitor performs no analysis, whereas in (b) the event monitor performs all the analysis. Case (c) is a hybrid between off-line and on-line analysis, in which the event monitor performs some analysis. The use of the results of the analysis as well as external attributes of a system such as performance, reliability, etc. determines how much analysis a monitor can perform on-line. In typical off-line usage (of the results), the monitor only performs necessary analysis such as compressing data and removing redundant observations so that the overhead cost of the analysis is minimized. In typical on-line usage, such as in active databases, the issue is to find a balance between the cost of monitoring and the cost of evaluating the database state. The less information that is carried with the event, the more the database state has to be evaluated to take some action that affects the database, and vice versa. In this Ph.D. thesis, on-line analysis and on-line usage of the result is emphasized.

Another view of event monitoring is depicted in Fig. 2.2 on the following page. Event composition is the process of collecting events (from the sources), analyzing them and signalling event subscribers when events occur, where each part (source, event monitor, subscriber) may reside in a separate address space that may be separated on different physical nodes in a distributed system.
2.1.1 Event Detection and Composition in Event Monitoring

A typical use of on-line analysis for on-line usage is detection of patterns in the observations. This pattern detection is denoted event detection in active databases, for example, HiPaC [CBB+89], Snoop [CKAK94], SAMOS [Gat94], ODE (e.g., [GJS93]) monitoring based on real-time logic (RTL) [LMK98]. Moreover, event detection is similar to Chronicle recognition [Dou96] in the sense that Chronicle recognition detects a pattern of occurrences. In active databases, the event composition denotes the process of performing pattern detection, since composite event occurrences are composed out of contributing event occurrences. The simplest pattern consists of the specification of a single event occurrence, in which analysis can be removed by letting the data collection notify actions directly when primitive events occurs. For example, in the REACH active database prototype (e.g., Buchmann et al. [BZBW95]), patterns consisting of single events are treated differently from patterns consisting of multiple events.

The purpose of event monitoring determines what is desirable to observe. In active databases, database events are monitored. If an event is detected, then its associated rules are triggered. These rules can be used to, for example, maintain consistency in a transaction. In this case, it is desirable to monitor updates of a database to check whether these updates leave the database in a state that does not violate any invariant such as “no salary may be greater than 1000.00$ per year”. The action taken can, for example, be to abort a transaction if it attempts to violate an invariant. Another example is testing, (e.g., Schütz [Sch94b], Beizer [Bei90]), where we want to show the presence of failures. For example, in structural testing [Bei90] the purpose is to discover faults relating to code structure. An example of fault is dead code, that is, code that will never be executed. To discover dead code, we need to instrument the code in such a way that the code that
has been executed can be observed. The code that has not been executed is potential dead code.

In a different context, such as electrical engineering, the concepts can be exemplified as follows. Assume that the temperature of some chemical process must be measured. Since we have thermistors, that is, resistors whose resistance varies in a predictable way (close to linearly) with the temperature within a temperature range, then we can use a thermistor as a sensor. By applying voltage over the thermistor, it is possible to measure the current. When the temperature rises, the resistance decreases and the current increases. The monitor in this case is the device measuring the current and translates that into temperature readings. The data passed between the sensor and the measuring device is the current.

2.1.2 Event Subscription

A common feature in event monitoring is event subscription. For example, in an active database (e.g., [CM94, GJ92b, Gat94]) rules subscribe to events. In real-time systems, tasks can be triggered by events that they subscribe to.

In general, no a priori knowledge of the number of subscribers is necessary allowing flexibility in subscription. The advantage is that this scheme allows post-development changes to take place. The disadvantage is that it is impossible to estimate the resource requirements (in terms of processor time and memory) of event subscription unless it is constrained in some way, for example, by enforcing a maximum number of subscribers.

2.1.3 Probe Effect and Observability

An important issue is that observation affects the monitored object and the sensor. In terms of software, inserted code (used as sensors) affects, for example, performance. In terms of electrical engineering, the voltage used sends a current through the thermistor, heats it up and, thus, we may read a higher temperature than the temperature of the monitored object (e.g., air, a processor). To be more precise, in computer systems, monitoring affects the behavior of a monitored object. For example, a monitored task has longer (possibly unpredictable) response times due to the required instrumentation; a task’s execution can be indirectly affected by execution of the monitor. The earliest work discussing this effect in computer science and defining it as the probe effect was carried out by Gait [Gai85]. The probe effect is the difference
in behavior between a monitored object and its unmonitored counterpart. This probe-effect is denoted *intrusion* by Schroeder [Sch95a].

An issue related to the probe effect is observability. Observability is a major issue in the testing of real-time systems according to Schütz [Sch94b, Sch95b]. In this research area, observability is defined as the system’s capability to facilitate event monitoring for testing purposes. If we have a probe effect, then this might invalidate tests. For example, given a system that is instrumented for testing purposes and this instrumentation is removed before the system is deployed, then we introduce a probe effect. Whether this probe effect is significant or not depends on the purpose of the tests. For example, if we test some external property such as performance, then the performance will change after removing the instrumentation. In contrast, if we test logical correctness of a non-concurrent system, then the instrumentation should not introduce a probe effect given that it does not affect this logical correctness by, for example, introducing logical faults.

Since this Ph.D. thesis is addressing a more general area than testing, *observability* is defined as a system’s capability of facilitating event monitoring for all desirable purposes. This is a generalization of the aforementioned definition in testing. For example, if a system administrator wants to view the behavior in terms of throughput of a system, then it has to be observable in terms of throughput.

### 2.1.4 Synchronous and Asynchronous Monitoring

Chodrow et al. [CJD91] differentiate between synchronous and asynchronous monitoring. The key distinction is whether the monitor is located within the monitored object itself. In *synchronous* monitoring, the monitor as well as the sensors are (in) the monitored object itself. That is, the monitored object and the monitor itself competes for the same resources. In contrast, in *asynchronous* monitoring the monitor resides outside the monitored object. The overhead caused by synchronous monitoring may aggravate the probe effect, since the resources executing the monitored object do not only execute the sensors but also the monitor. In contrast, in asynchronous monitoring there is a greater possibility to control the overhead, for example, by off-loading the monitor to additional hardware in the system (e.g., Mellin [Mel98a]).
2.1 Structure of Event Monitoring

2.1.5 Hardware, Software, and Hybrid Event Monitors

Hybranietz et al. [HW90] differentiate between hardware monitors implemented using hardware only, software monitors using software only and hybrid monitors that are implemented using a combination of both. Hardware and hybrid monitors perform asynchronous monitoring, since the monitor uses different hardware with respect to the monitored object. Hybrid monitors are considered better than pure hardware and software monitors, because they combine the flexibility of software monitors with efficiency of hardware monitors (in Table 2.1 on the following page). This improved efficiency is due to the fact that the overhead of executing the sensors in hardware monitors is less than in software monitors. A drawback of hardware (and hybrid) monitors compared to software monitors is the need to build special purpose hardware, but this can normally be kept to a minimum in hybrid monitors.

2.1.6 Resource Requirements of Event Monitoring

The resource requirements of monitoring are twofold: firstly, the size of the filtered event log is important since memory, disk etc. are required to store it; secondly, the overhead cost in terms of processor time can be significant. In this section, the first requirement and the instrumentation overhead of the second requirements are addressed.

The logged events must often be stored, at least, until they are analyzed. The storage can be a log of all events (i.e., unfiltered log) or it can be optimized, for example, by compressing the data or using counters to observe how many times a particular statement has been executed rather than logging each occurrence. Even though we can optimize the storage requirements, it is important to keep the size of the filtered event log minimal to reduce: (i) the necessary resources such as memory; and (ii) the time it takes to analyze the history which is dependent on the size. The size of this log is dependent on the abstraction level of the events. If the abstraction level is low, more events must be logged to extract the necessary information from the system, compared to a high abstraction level (in Table 2.1 on the following page). For example, assuming that we want to monitor the execution of code, then if a hardware bus is monitored then all signals (bus

---

3 In some cases hybrid monitors require only general purpose hardware. For example, in modern dual-processor systems it is possible to dedicate one processor for monitoring activities performed on the other processor.
events) must be logged; however, if a high-level language is monitored by a software monitor, then fewer events are necessary to log. In the latter case, it is possible to insert sensors between each statement that register the occurrence of completing one statement and starting another statement. Since statements are translated into several machine instructions, the number of observations necessary to reconstruct what has been executed is larger for hardware monitoring compared to software monitoring.

The overhead cost of instrumentation is less in hardware and hybrid monitoring than in software monitoring. The flexibility in terms of changing the instrumentation is greater in hybrid and software monitoring compared to hardware monitoring.

<table>
<thead>
<tr>
<th>Monitor type</th>
<th>Abstraction level</th>
<th>Size of filtered event log</th>
<th>Overhead of instrumentation</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Low</td>
<td>Large</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Software</td>
<td>High</td>
<td>Small</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Hybrid</td>
<td>High</td>
<td>Small</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of monitor types

2.1.7 Optimizations of Event Monitoring

It is possible to consider other forms of monitoring as optimizations of event monitoring. In many situations, it is possible to compact the filtered event log to contain only a counter representing the sum of the number of event occurrences. For example, this can be the case in some types of performance monitoring. A purpose of such performance monitoring can be to optimize application code by refining the most frequently called functions. Thus, by only counting the number of times each function is called, then it is possible to identify which part of the application where optimization improves performance in the best way.

Given the scenario in performance monitoring, it is possible to move the counter into the sensor. For example, the sensors have their own local variables that they update. Instead of signaling an event (function call started or function call terminated) each time it occurs, the event monitor can fetch the variables from sensor when needed.
2.2 Management of Event Monitoring

Event monitoring follows parts of the steps of managing traditional (off-line) monitoring. The traditional 8 steps summarized by Snodgrass [Sno88] (emphasized below) in the monitoring of computer systems for off-line usage occur during the following four phases:

1. Design and implementation of a system: *(a) Sensor configuration:* This step involves deciding what information each sensor records (and possibly reports) and where each sensor is located, if required; *(b) Sensor installation:* The sensors are coded (or built in hardware) and inserted in the correct location in the monitored object (originally called the “subject program” by Snodgrass). Provision must be made for storage of the collected data in the sensor.

2. System setup: *Enabling sensors:* Some sensors are permanently enabled, either reporting monitoring data whenever executed (which is the case in event monitoring) or storing (and possibly reporting) these data, while others may be individually or collectively enabled, directly or indirectly by directives from the user.

3. Execution of a system: *Data generation:* The monitored object is executed, and the collected data are analyzed immediately or stored in main memory or on secondary storage for analysis later. No dynamic reconfiguration is generally possible during this phase, for example, it is not possible to change what the sensors should record during runtime (unless the sensors are designed to adapt to anticipated changes).

4. Post-execution analysis of the generated data: *(a) Analysis specification:* In most systems, the user is given a menu of supported analyses; sometimes a simple command language is available. *(b) Display specification:* The user is given a set of formats (either specified via a menu or a command language), ranging from a list of raw data packets to canned reports or simple graphics. *(c) Data analysis:* Data analysis normally occurs in batch mode after the data have been collected due to the often high computational cost of the analysis. *(d) Display generation:* Usually this step occurs immediately after data analysis, although a few monitoring packages allow the analyzed data to be displayed at a later time.
As a result of a declarative (relational) approach for specifying monitors, Snodgrass [Sno88] reduced these steps to the following 5 steps: (1a) Sensor configuration, (1b) Sensor installation, (4a) Analysis specification, (4b) Display specification, and finally (3) Execution (where the missing steps of enabling sensors, data analysis, and display generation are performed automatically in the Execution step). Snodgrass used historical databases and a relational query language TQuel as a basis for this simplification. This declarative approach makes monitoring easier to maintain and extend compared to the traditional approach. The major drawbacks mentioned by Snodgrass are that:

1. the queries are specified before the events (or data) are collected. This a priori knowledge is not always available.

2. the complexity (or asymptotic efficiency) of execution of the relational monitor makes it inefficient compared to traditional procedural approaches. In a practical test [Sno88] the performance of this approach in TQuel (including optimization of queries) was two orders of magnitudes worse than a traditional procedural approach (in LISP).

Event monitoring follows the steps defined by Snodgrass with one exception: event monitoring for on-line usage does not concern displaying data in most application areas such as active databases, real-time systems etc. The reason is that the main purpose is to analyze the filtered event log to take appropriate action rather than, for example, to show the presence of faults in order to remove them. Also, all existing approaches to event monitoring are declarative since it is advantageous in terms of maintenance.

2.2.1 Instrumentation of a Monitored Object

Sensor configuration and installation are referred to as instrumentation. In Section 2.1.6 on p. 17, it is argued that the overhead of hardware and hybrid monitoring is low. In this section, two examples of hybrid monitors with low computational cost are briefly presented to substantiate this claim. Additionally, a brief discussion on how to automatically instrument applications are addressed.
Hybrid Monitoring Solutions

In the INCAS project, Hybranietz et al. [HW90] introduced a bus-snooping hybrid monitor for monitoring distributed systems. Their solution causes only 1% overhead in the monitored object while monitoring up to 1300 events per second, since the monitor resides on a dedicated processor attached to the same bus as the application processor. Both processors are general-purpose hardware. The sensors are inserted as code visible on the data bus so that the dedicated processor, which is snooping the bus, can observe the occurrences. Jahanian et al. [JRR94] employed the INCAS system for time constraint monitoring, in particular, deadline monitoring. In other words, they investigated monitoring of task completion with respect to the allotted deadline of each task.

Gorlick [Gor91] addresses the problem that hardware optimizations lead to less data being visible on the data bus. For example, the use of caches reduces the usage of the data bus. His solution is to use a co-processor instead of a bus-snooping processor. He claims that a co-processor-based solution can monitor events with a higher frequency (or shorter minimum interarrival times) than a bus-snooping-based solution.

Object-Oriented Database Management Systems

In an object-oriented database management system (DBMS), classes are made persistent either by inheriting from a persistent class hierarchy or by attaching objects at run-time to objects handling persistence.

According to Buchmann et al. [BZBW95], there are four ways of instrumenting an object-oriented DBMS using software sensors:

(i) The compilation process can be altered to instrument methods and procedures; a difficult task because compilers vary, language specifications change, and erroneous inputs must be handled. However, both DBMS operations and user specified methods/procedures are handled.

(ii) Call-back hooks in the DBMS can be used; however, experience has shown that no closed architectures have all appropriate hooks.

(iii) Each class in the class hierarchy is extended with an instrumented class, but the instrumentation will not be fully transpar-
ent as the user specified classes must inherit from the extended classes.

(iv) The existing classes in the class hierarchy can be changed; however, this can only be done in a public architecture where the source code is available and only DBMS operations can be handled.

The more flexible scheme of changing the compilation (i) is desirable as it was concluded [BZBW95] that extending the class hierarchy (ii) is feasible, but experience has shown that it often has the undesirable effect that all classes are persistent and all methods are instrumented.

2.3 The Notion of Time in Analysis Event Logs

“Tempus Frangit”

/Neil Gaimann

The view of time can differ in monitoring for on-line and off-line usage, since in on-line usage the event history up until now is known (in Fig. 2.3), whereas in off-line use the whole event history is known. This analysis for off-line use is akin to temporal problems in artificial intelligence, where, according to Galton et al. [GA01], it is assumed that the complete event history is known.

Moreover, assuming that the time moves from left to right depicted in Fig. 2.3, then in the analysis of event logs time can be viewed as linear, parallel, cyclical, left-branching, or right-branching. This view is used in temporal problems in artificial intelligence (e.g., McDermott [McD82]) such as planning. For example, in a planning problem several different possible solutions may be investigated, where each solution follows its own time. If these views of time are independent, then time is parallel. In contrast, if
these views of time are dependent, then they are either left-branching or right-branching.

The difference between left-branching and right-branching is whether possible solutions to the planning problem are investigated backwards (or left) or forward (or right) in time. In the left-branching view, at each decision point in a possible solution, the investigation is moved backward in time in order to find a solution. In contrast, in the right-branching view, this investigation is carried out forwards in time.

For example, Fig. 2.4 on the following page depicts reasoning in right-branching time under the following assumptions; there are three events (A, B, and C) that may occur in sequence (following the alphabetical sorting order). For example, “AC”, and “ABC” are possible solutions. Only the occurrence or non-occurrence of each event is considered in this example, where the occurrence is denoted with a circle. The question is what sequence gives the best result. For each possible occurrence or non-occurrence in this example, there is a branch in the line of reasoning to the right (the future). The planning is performed by going from some reference point in time and moving forward. For each decision, the planning follows the branches into the future.

In contrast, the reasoning in left-branching time looks backwards in time. This is depicted in Fig. 2.5 on the following page. The planning is performed by going from some reference point and backwards in time.

In contrast to planning, where different event sequences are analyzed to find a viable solution to a planning problem, event monitoring deals with matching actual event occurrences with some pattern. In this case, the view of time serves as an explanation of policies for detecting patterns. For example, assume that the pattern “either an A or B, followed by a C” is considered. An example of a right-branching policy is to only keep the most recent events; if “ABC” occurs, this matches with “L4” since “L5”-“L8” are thrown away when “B” occurs (more recently than “A”). In effect, the occurrence of “ABC” is the same as “BC”. In contrast, an example of a left-branching policy is to detect all possible sequences that terminate in the same way; if “ABC” occurs, then both “L6” (“AC”) and “L7” (“BC”) has occurred.

\[4\text{In reality, there are different assignments of time points to the events as well as different permutations of ordering of events; a fact that leads to more branches.}\]
Figure 2.4: Example of right-branching time

Figure 2.5: Example of left-branching time
Chapter 3

Event Monitoring in Distributed and Real-Time Systems

“Just as Einstein observed that space was not an absolute, but depended on the observer’s movement in space, and that time was not an absolute, but depended on the observer’s movement in time, so it is now realized that numbers are not absolute, but depend on the observer’s movement in restaurants.”

/Douglas Adams

Event monitoring in a real-time systems context must meet real-time requirements. Therefore, general aspects of real-time systems as well as specific aspects related to event monitoring are presented in this chapter. Since many real-time systems are distributed, monitoring of distributed real-time systems are briefly discussed in the following section.

Finally, a discussion of applications from a dependability perspective as well as an active database perspective is discussed. The dependability perspective is important, since many real-time systems must be dependable in some aspect, for example, medical equipment must continue to work even if something fails (safety) or a telephone switch must be highly available to customers. Further, the dependability community has a well-founded terminology edited by Laprie [L+94], in which applications of monitoring can be categorized.
Event monitoring is a major part of active (real-time) databases. Therefore, the monitoring in such architectures is introduced briefly. Important issues in event composition are illuminated.

### 3.1 Issues in Real-Time Systems

The major issue in real-time systems is that they must produce responses to external stimuli in a timely manner, as well as being logically correct. There are several definitions of real-time systems, such as [KV94]:

"The computer system must react to stimuli from the controlled object (or the operator) within time intervals dictated by its environment."

or [You82]:

"any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period"

Note that time constraints can be deadlines as well as delays (e.g., [BW89]).

A general view of (distributed) real-time systems by Kopetz et al. [KV94] is depicted in Fig. 3.1 on the facing page. A **real-time entity** is a physical part of the environment that the real-time system monitors or controls, for example, a temperature sensor or a fluid valve. A **representative** of a real-time entity is a part of the real-time system that observes or controls, respectively, the real-time entity. The former representatives can be called sensor representatives and the latter actuator representatives. A **computing element**, for example, a processor, handles the input from the sensors and triggers actions on the actuators. Time is a special entity that is necessary in any real-time system. A real-time system has at least one sensor, one actuator, and one computing element. These sensor representatives are an instrumentation of the environment (as the monitored object), where the real-time system monitors the behavior of this environment and takes appropriate actions (cf. Section 2.1 on p. 12).

To manage real-time systems, computations are divided into tasks. There are several definitions of a task. In this Ph.D. thesis, a **task** is an arbitrary computation that is executed in response to some stimuli (internal or external). During the execution, a task may interact with other tasks (e.g., by
3.1 Issues in Real-Time Systems

In contrast, tasks may have a more restrictive definition as in MARS [KDK+89, VK94]: a task first reads its input, processes the input, and then writes its output to the actuators. Fundamentally, by construction, this task definition can simplify concurrent programming and reduce it to a non-problem [KDK+89, VK94]. This more restrictive definition is subsumed by the former in the sense that a system that can handle the less restrictive task model can also handle the more restrictive task model.

3.1.1 Timeliness

Given this overall view, timeliness can be defined. Timeliness is defined as that given resources (i.e., sensors, actuators, and computing elements) in a system can complete a set of tasks such that they meet their time constraints. To ensure that timeliness can be achieved, timeliness is defined in terms of the necessary and sufficient requirements predictability and sufficient efficiency of algorithms executed by the tasks. Predictability in real-time systems is defined as that there is a limit on the resource requirements in terms of processor time, memory usage etc. of a task. Sufficient efficiency is defined as the part of timeliness, given a fulfillment of the predictability requirement, which ensures that available resources can guarantee that time
constraints of a set of tasks met. For example, if either an algorithm to perform something is unlimited in terms of the number of processing steps in an algorithm, or the available resources are inadequate to perform the tasks while meeting their time constraints, then the system cannot be timely. N.B. event monitoring is realized as a non-empty set of tasks and, therefore, this service must heed the requirements derived from timeliness.

There are basically two ways of managing timeliness requirements: either (i) by meeting them using scheduling or (ii) by relaxing the timeliness requirements (in case existing resources cannot be used to meet timeliness). Examples of (i) are: rate monotonic scheduling where seminal work was done by Liu et al. [LL73] that is now summarized by Klein et al. [KRP+93], and earliest deadline first scheduling (e.g., [LL73]). Example of (ii) is that Burns et al. [BW89] consider the definition by Young [You82] to be too broad, because the definition can be relaxed to allow a viable design for a real-time system. To allow relaxations, tasks are further divided into hard and soft tasks depending on the result of a missed deadline. In a hard real-time system, it is imperative that a task finishes before its deadline. In a soft real-time system, however, a missed deadline does not result in a catastrophic failure.

3.1.2 Event Shower Management

Management of event showers are important for (event monitoring in) real-time systems, since event showers may force a system to violate its timeliness requirements [KV94, p. 426, p. 456]. An event shower is a burst of events, often consisting of redundant information such as duplicate events representing the same real-world situation, that may cause an overload in a real-time system. These showers often occur in exceptional situations (fire, leakage, explosion, crash, etc.), where a number of nodes provide alarm information concerning a single cause. Some of this alarm information may be redundant or repeated and some of it may be multiplied by propagation.

No system can handle a completely random load. Aperiodic events, that is, events whose interarrival time is unspecified, are not tractable in a deterministic way in terms of resources needed to handle them. We can model these as sporadic events by constraining the environment to have a minimum interarrival time of events. Systems with sporadic events can be made tractable [KV94].
Further, when an alarm occurs the subsequent event generation can often be predicted to a significant extent. According to Kopetz et al. [KV94], functions can be realized by the designer to: (i) compact successive instantiations (repeated events) of the same alarm at the representatives; (ii) discard redundant events from different sources, either at the representatives or upon arrival at the computing element; and (iii) prepare communication and computing resources for the forthcoming shower. An example of such functions are found in the adjustable control flow filters proposed by Stankovic [Sta95].

Additional engineering measures can further improve the event handling capability. These measures have to do with load or flow control, concerned with regulating the flow of data from periphery (representatives) to the nucleus of the system (node). For example, if the real world events are bursty the transmission from the representative to the nodes can be smoothed, by spacing them by the equivalent of an average rate [KV94]. Smoothing is also known as transforming aperiodic events into sporadic events.

3.1.3 Declarative Monitoring in Real-Time Systems

The drawbacks discussed by Snodgrass [Sno88] (in Section 2.2 on p. 19) have no or less impact in a real-time system context. The need for a priori knowledge about the environment and the computer must be known in a real-time system, otherwise we cannot achieve timeliness. Therefore, the first drawback concerning a priori knowledge for non-real-time systems is, at least, not as severe for real-time systems compared to non-real-time systems. The second drawback that concerns the overhead cost of a declarative relational approach can be severe in a real-time system, since there is a difference in performance of two orders of magnitude in favor of non-declarative (procedural) monitoring. The performance and algorithmic time complexity of relational monitors is a critical issue, because the worst-case execution time of the monitor may be too pessimistic to be useful. That is, the time required to detect the occurrence of an event may have too long a duration.

The most severe problem with relational monitors is that the monitoring of complex event patterns uses the Cartesian product operator, which may be applied several times. Optimizations (or improvements) of the TQuel queries are available, but these only exist for a few of the possible expressions in TQuel.

\footnote{That is, redundant, repeated, or multiplied events.}
The relational approach suggested by Snodgrass has several advantages, but the disadvantage of algorithmic time complexity makes it virtually useless for event monitoring in real-time systems. However, by using less generic, but still powerful, languages based on operator grammars specifically designed for dealing with events, event composition can be made more efficient (cf. event contexts in Section 4.5 on p. 51). In contrast, these languages are meant for on-line analysis of the monitored data for on-line usage. Hence, step 4 (analysis) in Section 2.2 on p. 19 is not executed during post-execution. Moreover, the purpose is not to give a user a good view of monitored data, but to provide subscribers with required information.

3.1.4 Probe Effect in Real-Time Systems

The probe effect (in Section 2.1.3 on p. 15) consists of two parts [Mel98a]: (i) the unpredictability of the overhead of the event monitor, and (ii) the removal of the instrumentation from a monitored system to, for example, enable delivery to customers and thereby generate the unmonitored counterpart. By using asynchronous monitoring, part (i) can be handled by differentiating between sensors, which can be designed to be predictable while only introducing a small overhead in the monitored object (in comparison to the complete monitor), and the data collection and analysis that can be unpredictable. In this case, sensors are introduced into (tasks of) the monitored object, whereas the data collection and analysis are executed by tasks outside the monitored object. However, this does not solve the problem for on-line usage, since it requires the data collection as well as the analysis to be predictable. As mentioned briefly, this problem is the focus of this work.

The difference in the behavior of a monitored system and its unmonitored counterpart can only be avoided by leaving the sensors and event monitors in the operational system. However, this is only imperative in hard (and best-effort\(^2\)) real-time systems [Sch94b, Sch95b], because observability must be ensured. In this Ph.D. thesis the probe effect must be avoided, because a predictable service is desirable. In non-hard real-time systems it may be sufficient to compensate for the probe effect by getting get close to the required observability (e.g., Compensating for intrusion by Gannon et al. [GWA+94]).

\(^2\)Best-effort real-time systems are assumed to guarantee hard deadlines.
3.2 Issues in Distributed Systems

Monitoring in distributed systems must meet general and specific requirements of distribution. In this Ph.D. thesis, a distributed system is viewed as loosely coupled clusters of computing elements (nodes) that share neither memory nor global clock (e.g., [BW89]). In contrast, a cluster of computing elements are assumed to share memory and clock. Such a cluster of computing elements is denoted a physical node.

A distributed system can be either homogeneous or heterogeneous. In a homogeneous system, all nodes share the same data representation and data alignment and normally execute the same operating system.

The major issue concerning monitoring, as well as most distributed real-time applications, are time and order of events. That is, when did something actually occur and in which order did something occur. Since we lack a global state, there can be temporary discrepancies in the distributed state. Further, the overhead costs of communication links and potential uncertainties in communication delays can aggravate the probe effect. For example, the monitoring of an application can delay application-related communication over the network.

Since distributed systems are potentially fault-tolerant, there are several application areas for monitoring. For example, given that there is one main computer and a backup computer and that the backup should take over the service when the main computer fails, then the backup needs to monitor the behavior of the main computer in such a way that the backup can takeover. An example of such monitoring can be to periodically send a probe and await a response. If no response is received within a time limit, then it is assumed that the main computer has failed.

3.2.1 Distributed Real-Time Systems

It is strongly desirable to support a distributed real-time system by reliable real-time communication in order to guarantee response time of real-time remote requests. This is desirable since it simplifies the management of timeliness.

Verissimo [Ver94b] claims that given a bounded transmission delay and bounded omission degree it is possible to ensure an upper bound on the interval between the sending of a message over the network and the delivery of the message to the recipient tasks. This may include multicasts. This
guarantee requires that traffic patterns are known and latency classes on
the network are defined. It is possible to ensure this due to that omission
failures on networks are rare and of a bursty nature.

### 3.2.2 Global Time Base and Global Ordering

Global ordering of events and requests must be preserved in a distributed
system in order to prove global properties about the system. Without global
ordering, it is impossible to ensure that requests are served in the order they
were made. Additionally, it impossible to tell in which order two (causally
dependent) events occurred. In a distributed real-time system, a global
timebase is desirable to support, otherwise according to Veríssimo [Ver94a,
Ver94b] it is impossible to (i) synchronize the triggering of actions at two
different nodes, (ii) do distributed logging and, hence, global event detection.

In distributed systems, the $\delta_t - \text{precedence}$ relation (i.e., $\delta_t$ is the
minimum real time interval for causal relation to be generated [Ver94a]) is the
criterion for potential causality. The $\delta_t = 2g$ given a clock granularity $g$
[Ver94a], where $g$ must be larger than or equivalent to the bounded clock
precision $\pi$ (cf. clock-synchronization algorithms [Chr89, CGG94]).

### 3.3 Event Monitoring for Fault Management

In order to explain how event monitoring can be used for fault manage-
ment, the fault pathology needs to be explained. According to Laprie et
al. ([L+94]), a fault is the hypothesized cause of an error; an error is the
incorrect state a component enters when a fault occurs. Moreover, a failure
is a deviation from the expected behavior of the component, due to an error.
A failure in one component is considered a fault in another component.

In fault management, event monitoring can be used for fault tolerance
and fault removal that are two of four overall methods to manage faults pre-
sented by Laprie et al. [L+94]. The other two methods are fault prevention
and fault forecasting; in these two methods there is no obvious need for event
monitoring, since fault prevention addresses conditioning a system in such a
way that a particular fault can never occur and fault forecasting emphasizes
the prediction of faults between different increments of a system during its
development to control the development process.
3.4 Active Real-Time Databases

Event Monitoring for Fault Tolerance

Fault tolerance implies that redundancy is added to a system to tolerate faults. This redundancy can be added to space (e.g., additional computing elements), to time (e.g., additional request/reply pairs to compensate for omissions), or to information (e.g., check sums used in communication). In particular, redundancy in time can employ event monitoring to recover from failures in a subsystem. For example, in a distributed system requests (for a service) sent over the network may be lost. One way of detecting omissions is to wait for a timeout relative to the initiation of each request. In other words, for each request monitor for either the completion (a request is followed by a reply) or the omission (a request is followed by a timeout). Additionally, it is possible to detect temporal failures such as missed deadlines etc.

Effectively, this means that event monitoring can be used for passive redundancy. Passive redundancy [KV94] is invoked when a failure occurs in a subsystem. It is also known as dynamic replication [BW89], since the behavior changes when an error occurs. However, event monitoring is not as useful for active redundancy. In active redundancy [KV94] (also known as static replication [BW89]), redundant resources are used under the assumption that failures occur all the time.

Event Monitoring for Fault Removal

Event monitoring can be used to achieve observability (in Section 2.1.3 on p. 15). Therefore, given that there is event monitoring in the system for application purposes, then it is possible to use event monitoring for testing purposes with an acceptable penalty in terms of computational cost.

Further, event monitoring for off-line and on-line usage can be used together (cf. Section 2.1 on p. 12). For example, if request/reply pairs are logged and event monitoring aggregates this information anyway, then the filtered event log for testing purposes can be reduced in terms of number of logged events; thereby, reducing the cost of analyzing the filtered event log.

3.4 Active Real-Time Databases

The research field of active databases is active where early work was presented in the context of HiPAC [DBB+88], ETM [KDM88], and POSTGRES [SHH87]. These can be classified as event-triggered systems (although not
necessarily real-time systems). For example, the DeeDS prototype adopts the active database idea since it conforms to the event-triggered real-time design paradigm [AHE+96].

The major idea is to add reactive mechanisms to the database as ECA-rules. An ECA-rule is an association of an Event, a Condition, and an Action. When a specified event occurs, the associated rules are triggered and their conditions evaluated. For all rules that have been triggered, if their conditions are true their associated actions are performed.

### 3.4.1 Transactions and Coupling Modes

In this Ph.D. thesis, the definitions of Gray and Reuter are used [GR94] concerning transactions:

“A transaction is a collection of operations on the physical and abstract application state.”

In contrast, other definitions of transactions are according to Gray and Reuter: (i) the request or input message that starts an operation, which they denote *transaction request/reply*, and (ii) the program(s) that execute(s) an operation, which they denote *transaction program*. An operation is a function that is meaningful in some context and desirable to perform, for example, withdraw money from an account.

The relation between tasks and transaction are as follows: a transaction is executed by one or more tasks; a transaction program may consist of one or more tasks. A task is assumed to be able to join, leave, and resume a transaction. For example, a task can leave a transaction to serve a client (task) in another transaction, then it can resume the old transaction.

A transaction should have four properties [GR94]: *atomicity*, that is, all changes to the state made by a transaction are atomic; *consistency*, that is, since a transaction should be a correct transformation of the state, then the changes may not violate any integrity constraints; *isolation*, that is, concurrent tasks in different transactions should not perceive each other’s intermediate state changes; and *durability*, that is, if a transaction is completed successfully, then changes to the state survives failure.

For each part of an ECA-rule, there is a corresponding mechanism that computes something. To detect events, event monitoring is used. A rule manager is responsible for evaluating what rule is triggered by the detected events, evaluate their conditions. For each condition that holds true, its
corresponding action is submitted for scheduling. These mechanisms may be realized as one or more tasks.

Each of these mechanisms can be evaluated in various ways with respect to transactions. This is referred to as coupling modes, that is, which mechanisms are executed together within a transaction. In the HiPAC project [CBB+89], seven different coupling modes were identified. One of these coupling modes was refined by Branding et al. [BBKZ94]. For example, condition evaluation can take place immediately in the same transaction as the triggering event, and the action can be executed in a different (detached) transaction with respect to the condition evaluation. That is, a coupling mode defines how event monitoring, condition evaluation, and action execution are performed with respect to transactions. The example is an immediate-detached, since the coupling between event monitoring and condition evaluation is immediate, whereas the coupling between condition evaluation and action execution is detached.

There are seven combinations defined by Chakravarthy et al. [CBB+89]:
(1) immediate-immediate, (2) immediate-deferred, (3) immediate-detached, (4) deferred-deferred, (5) deferred-detached, (6) detached-immediate, and (7) detached-detached. Deferred implies that the mechanism is executed at the end of a transaction. Branding et al. define variations of detached coupling modes.

In a real-time system, immediate and deferred coupling modes are problematic, since they may introduce a significant difference between the worst-case execution time (possibly unpredictable) and the average execution time [Eri97]. Additionally, synchronous monitoring (in Section 2.1.4 on p. 16) of events may also introduce varying (and possibly unpredictable) execution time overhead to a transaction. Therefore, it is argued that in an active real-time database, these coupling modes should be avoided [Han99]. Also, as discussed in Section 3.1.4 on p. 30 it is argued that synchronous monitoring should be avoided to manage the probe effect.

### 3.5 Patterns of Events and Transactions

In active databases, it is desirable to detect patterns consisting of multiple events. These are called composite events, for example, in Snoop [CKAK94], SAMOS [Gat94], etc. One problem emphasized by Buchmann et al. [BZBW95] is that events belonging to a composite event may not stem
from the same transaction. As long as constituent events in a composite event stem from the same transaction, it is possible to stop monitoring for the pattern when the transaction is terminated. However, if the contributing events stem from different transactions, then they must be saved after the transaction in which the contributing event was generated has terminated. According to Buchmann et al., we need some (temporal) validity interval in which a pattern is valid. Given such a validity interval, then it is possible to remove events to save resources. No provision is made for how to define this validity interval by Buchmann et al.

The reason why we cannot always monitor for patterns in a transaction can be exemplified as follows. Assume that we are looking for a sequence of events, for example, open door followed by a close door which are sensed by real-time entities and delivered to computing elements via sensor representatives. If these real-world events are monitored in a transaction, then the following scenario may occur: The close door event may not occur, because the door or the sensor breaks. In this case, the transaction does not complete. Since the representation of the open door event is locked by a transaction, then no other transaction can access it.

The two classical problems of long-running transactions and hot spots [GR94] are faced in event monitoring within the scope of transactions. In the aforementioned example, a transaction may not complete due to failures; a significant problem if the events are hot spots (that are accessed by many transactions). Further, if we monitor a complex pattern involving several contributing events, then these events may be locked for too long a period of time before they are released. This latter problem relates to long-running transactions.

### 3.6 Event Monitoring for Real-Time Systems

Milne et al. [MNG+94] have employed Chronicle recognition that is a form of event monitoring where the analysis is based on simple temporal networks. Simple temporal networks by Dechter et al. [DMP91] have been used in artificial intelligence to solve planning problems by temporal constraint satisfaction. Dousson et al. [DG93, Dou96] introduced temporal constraint satisfaction of simple temporal networks as a way of analyzing the filtered event logs instead and named it chronicle recognition. Chronicle recognition is addressed in detail in Ch. 4.
A similar kind of analysis was introduced by Chodrow et al. [CJD91] based on real-time logic expressions [JM86, JMS88]. Briefly, a real-time logic expression is a boolean expression of relations, where each relation defines a temporal relation between two event occurrences of a historical order. For example, it is possible to express that the minimum allowed interarrival time between two events are of $D$ duration. Recent work has been performed by Liu and Mok [LM99, LMK98, ML97a, ML97b].

The difference in these approaches is the purpose of the analysis. The purpose of the work by Dousson is to reason about events in the environment. The reasoning follows a right-branching view of time (in Section 2.3 on p. 22). A simplified explanation is that there is one line of reasoning going on at the same time. This line of reasoning may have different branches representing a possible scenario that is occurring.

The purpose of the monitoring of real-time logic expressions has been to monitoring internal time constraints. Since computers can support meaningful ordering of events, then this has been employed in this analysis. Their analysis supports a parallel view of time and a special case of right-branching view of time. In this special case of right-branching time only the most recent branches are kept and the others are discarded.
Chapter 4

Event Composition: Syntax, Semantics, and Processing

“In order for one to deviate successfully, one has to have at least a passing acquaintance with whatever norm one expects to deviate from”

/Frank Zappa

In this chapter, the syntax, semantics, and pragmatics (processing) of event specification languages for event composition is presented to provide a sound basis for the addressed problem(s) in this Ph.D. thesis. Event specifications address how a filtered event log is analyzed in order to obtain higher-level (more abstract) event occurrences possibly consisting of other event occurrences. N.B. the presentation of the syntax, semantics, and pragmatics is not presented in a strict model-theoretic manner as suggested by, for example, Chang and Keisler [CK90], Manzano [Man99], Marker [Mar02], since processing is emphasized in this work, not the semantics of event specification languages. A looser form used by, for example, Galton et al. [GA01] and Chakravarthy et al. [CM94, CKAK94] has been adopted instead.

This chapter is based on an evaluation of key languages from the active database, real-time systems as well as artificial intelligence domains. In particular, it presents Solicitor [Mel98a] whose semantics is refined in Ch. 6. This evaluation emphasizes the processing of specified event types in terms of event composition processing, since it is a necessary requirement in critical
real-time systems to know that algorithms have bounded resource requirements (as discussed in Section 3.1 on p. 26, in particular Section 3.1.1 on p. 27).

The evaluation is not presented in a formal framework such as the evaluation by Motakis et al. [MZ97], or a structured and systematic framework such as the evaluation by Zimmer and Unger [ZU99]. For a more in-depth comparison in terms of semantics of active databases, the interested reader can read these publications. The result of these evaluations is that the event specification languages in active database approaches are similar and even equivalent in many aspects. In contrast, this comparison presented in this Ph.D. thesis also includes formal approaches not addressed by Motakis et al. and Zimmer et al. as well as relevant approaches outside the area of active databases.

An event specification language can be used to define meaningful event occurrences. Event occurrences are categorized by event types. An event type denotes event occurrences that have the same characteristic behavior (possibly expressed in terms of other event occurrences). As in active databases, event types are either primitive or composite [DBB+88, CBB+89, CM94]. Primitive event types are pre-defined events in the monitored object denoting that something important has occurred. For example, in an active database the request to commit transactions is an important event type, since it is needed to process deferred actions (in Section 3.4.1 on p. 34). This definition of primitive event types encompasses more precise definitions such that a primitive event type denotes a change in a condition [Pet00], where the condition is a part of a system requirements specification. In contrast to primitive event types, composite event types are defined by an event expression, where an event expression is a pattern that defines how event occurrences of specified event types can be combined. These patterns are defined using event operators that constrain how event occurrences can be combined. Examples of event operators are sequence and conjunction. Briefly, a composite event occurrence terminates when there is an occurrence of each contributing event type meeting the constraints of the event expression defining the composite event type. Primitive event occurrences are assumed to be atomic and instantaneous. This is similar to Chakravarthy et al. [CKAK94], who state that all event occurrences are atomic and instantaneous.
4.1 Addressed Prototypes and Event Specification Languages

In this chapter, the following event specification languages are addressed:

- Event specification languages in active database prototypes:
  - Snoop language stemming from the Sentinel active database prototype [CM94, CKAK94] is a main focus and it introduced the important concept of event contexts. It is the source of many other event specification languages such as GEM [MSS97] and the active database prototype REACH [Buc94, BZBW95]. Snoop and the event specification language REACH are refinements of the HiPAC active database project [CBB89]. Formalizations of the Snoop language have been addressed by Chakravarthy et al. [CM94, CKAK94, CYY98] and Adaikkalavan and Chakravarthy [Ada02, AC02] as well as Galton et al. [GA01].
  - SAMOS [Gat94, GGD94, GD93, GD92] is an active database prototype developed at the University of Zurich, Switzerland. Its event specification language is based on Petri-nets and it supports parameterized event types based on transaction (identities) and user (identities). For example, they can allow that occurrence of contributing event types belong to different transactions or require the occurrences to belong to the same transaction.
  - ODE [LGA96, GJS93, GJM93, GJ92b, GJ92a, JMS92] is an active database prototype developed at AT&T. Their event specification language COMPOSE is based on finite state automata.
  - Event Pattern Language (EPL) [MZ97] is an event specification language developed for deductive databases.

- Event specification languages from real-time systems:
  - Event control based on temporal calculi for communicating systems (TCCS) [BG97] is an interesting formalization of event detection.
  - Interval-based event algebra for restricted event detection [CL03] is a recent and interesting approach that attempts to maintain algebraic properties of event specification languages while addressing real-time systems.
Monitoring of (satisfaction or violation of) real-time logic expres-
sions [LM99, LMK98, ML97a, ML97b, CJD91] is a formalization
of event detection developed in real-time systems.

- Artificial intelligence:

  - Chronicle recognition [DGG93, Dou96] is a formalization of event
detection developed in the scope of artificial intelligence.

The event specification languages in these domains are similar. From a
usage perspective, the major difference is that event specification languages
for databases are based on transactions and are usually a part of their rule
specification language, whereas the languages in real-time systems and arti-
ficial intelligence are not based on transactions and are designed to be free
specification languages. N.B., these differences are not of major consequence
to our work.

4.2 Basic Formal Definitions

The basic formal definitions in this section are used throughout this work.
The basis for the notation is found in Appendix A. In the same way as
Galton et al. [GA01], a discrete time base is assumed where \( t \in \mathbb{N} \) denotes
a time point. The expression \([t, t']\) denotes an interval such that \( t \leq t'\), and
\( \text{now} \) represents the current time (cf. Section 2.3 on p. 22).

The symbol \( E \) denotes an event type. The predicate \( \text{prim}(E) \) is true if
and only if the event type \( E \) is primitive, otherwise it is false. The symbol \( T \)
denotes a duration where \( T \in \mathbb{N} \). Finally, the symbol \( Q \) denotes a predicate.

4.3 Syntax

The syntax defined here emphasizes event expressions. In an event specific-
cation language, there can be other syntactical constructs such as definition
of subscription by recipients (e.g., \( \textbf{R1: ON event IF condition DO action} \)
is an example of a subscription in an active database where the rule R1
subscribes to the \textit{event}); however, these are not necessary for our work.
4.3 Syntax

4.3.1 Abstract Event Expression Syntax

As mentioned, an event expression consists of event operators applied to (contributing) event types. For example, \(E_1; E_2; E_3\) is an event expression allowing a sequence of an \(E_1\) event occurrence followed by an \(E_2\) occurrence and an \(E_3\) occurrence. Each event operator defines the constraints that event occurrences matching a pattern must fulfill. For example, in a sequence (‘;’) an occurrence of the event type of the left operand must precede the occurrence of the event type of the right operand.

**Definition 4.1**

**Basic event expression syntax:**

Let \(E_P \in \{E|\text{prim}(E)\}\) in \(E ::= E_P \mid E;E \mid E\lor E \mid E\land E \mid N(E, E, E) \mid A(E, E, E) \mid A*(E, E, E) \mid P(E, [T], E) \mid E+T \mid E! \mid E[Q]\)

The abstract syntax in Def. 4.1 is right associative and stands for primitive event \((E_P)\), sequence (\(;)\), disjunction (\(\lor\)), conjunction (\(\land\)), non-occurrence (\(N\)), aperiodic expression (\(A\)), cumulative aperiodic expression (\(A*\)), periodic expression (\(P\)), relative temporal expression (\(+T\)), termination of (\(!\)), and logical occurrence (\([Q]\)). The semantics of the operators are formally defined in Defs 4.3 to 4.11 on pp. 47–49. In addition to the basic syntax, it is possible to assign attributes of the operator enclosed in angle brackets (e.g., \(E_1;(\text{context: chronic}\)le)\(E_2\) means that the event context is chronicle). It is also possible to check equivalence of parameters. For example, \(E_1;(\text{ptype: tid})E_2\) means that the attribute \(\text{tid}^1\) should be equivalent for the contributing event types in order for a match to occur. Further, attributes can be assigned values, for example, \(E = \text{tempReading}[^{\text{temperature > 95}}}^{\text{(criticality := high)}}\) means that \(E\) is a critical event type that occurs when a \(\text{tempReading}\) event occurs that has a temperature above 95 degrees Celsius. The attributes can also be tested in predicates, for example, \(E' = E[^{\text{span}(E)} = 0]\) meaning that an \(E'\) event occurs if and only if an event of event type \(E\) with zero duration has occurred. The complete event expression syntax of Solicitor is in Def. B.1 on p. 346.

The existing attributes in Solicitor used throughout this work are summarized in Table 4.1 on the following page. The columns are: “Attribute”.

---

^1It is assumed that \(\text{tid}\) means transaction identifier.
the name of the attribute, “Symbol”, the symbol used in an expression, “Assignable?” , “Y” if it is assignable by ‘:=’ and “N” if it is not, and “Comment”, is an additional comment about the attribute.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Symbol</th>
<th>Assignable?</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>context</td>
<td>N</td>
<td>Used to specify the context that an event operator works in.</td>
</tr>
<tr>
<td>Criticality</td>
<td>criticality</td>
<td>Y</td>
<td>Used to specify the criticality of the event occurrences of that event type (in Section 4.10 on p. 85).</td>
</tr>
<tr>
<td>Expiration</td>
<td>expire</td>
<td>N</td>
<td>Used to specify expiration of event composition of an initiated but non-terminated composite event occurrence (in Section 4.5.6 on p. 62 and Ch. 6).</td>
</tr>
<tr>
<td>Parameterization</td>
<td>ptype</td>
<td>N</td>
<td>Used to specify event type parameterization based on event type parameters.</td>
</tr>
<tr>
<td>Span</td>
<td>span</td>
<td>N(^{a})</td>
<td>Used to obtain the interval throughout which an event occurrences has occurred(^{b})</td>
</tr>
<tr>
<td>Type</td>
<td>type</td>
<td>N</td>
<td>Useful to check the event type of the terminating event occurrence in a disjunction to distinguish between different alternative termination possibilities.</td>
</tr>
</tbody>
</table>

\(^{a}\)Implicitly assigned during event composition.  
\(^{b}\)In other work, for example, Snoop [CM94, CKAK94] the tocc attribute returns the termination time.

### 4.3.2 Evaluation of the Abstract Event Expression Syntax

In Table 4.2 to 4.3 on pp. 45–46, a summary of the evaluation of the event operators in the abstract syntax are presented. The event operators are evaluated with respect to the existence of such an operator in the addressed event specification languages. The rows indicate the operator and the columns in-
dicate the language. “N/A” stands for “not applicable” meaning that there is no such operator in the language and it is not “possible to express” in the language. The “PTE” means possible to express and “SIM” means similar constructs exists. That is, for “PTE” using the semantics of the available operators in the language in question, then it is possible to express the event operator (or at least a more restricted form); this issue is addressed in this chapter.

As can be seen, the event operators in the abstract syntax have equivalent operators in existing languages. However, no language covers all event operators. The reason that Solicitor [Mel98a] lacks the cumulative aperiodic and the periodic event operators is that the former introduce unpredictable resource requirements and the latter is considered to be a scheduling specification; the scheduling specification should be part of a system specification, but not necessarily the event specification language of a system specification.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Solicitor</th>
<th>Active Database Prototypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>;</td>
<td>;</td>
<td>Snoop</td>
</tr>
<tr>
<td>Conjunction</td>
<td>∆</td>
<td>∆</td>
<td>and</td>
</tr>
<tr>
<td>Disjunction</td>
<td>∨</td>
<td>∨</td>
<td>or</td>
</tr>
<tr>
<td>Relative temporal</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Non-occurrence</td>
<td>N</td>
<td>N</td>
<td>Not</td>
</tr>
<tr>
<td>Aperiodic</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Cumulative aperi-</td>
<td>A⁺</td>
<td>N/A</td>
<td>A⁺</td>
</tr>
<tr>
<td>odic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Periodic</td>
<td>P</td>
<td>N/A</td>
<td>P</td>
</tr>
<tr>
<td>Termination of</td>
<td>!</td>
<td>!</td>
<td>N/A</td>
</tr>
<tr>
<td>Logical event</td>
<td>E[P]</td>
<td>E[P]</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of evaluation of abstract syntax for active database prototypes

4.3.3 Examples of Primitive Event Types

Primitive event types are dependent on the system in which the monitored object is executed (e.g., Mellin [Mel98a]). According to Buchmann et al. [BZBW95], the set of primitive events in an active object-oriented database management system must consist of: arbitrary method invocation events that
Table 4.3: Summary of evaluation of abstract syntax for formal approaches

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Solicitor</th>
<th>Formal Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>;</td>
<td>.</td>
<td>TCCS&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Δ</td>
<td>?</td>
<td>RTL&lt;sup&gt;b&lt;/sup&gt;, CR&lt;sup&gt;c&lt;/sup&gt;, CL&lt;sup&gt;d&lt;/sup&gt;</td>
</tr>
<tr>
<td>Disjunction</td>
<td>∨</td>
<td>+</td>
<td>PTE, PTE, ∨</td>
</tr>
<tr>
<td>Relative temporal</td>
<td>+</td>
<td>.ϵ(▵).τ</td>
<td>PTE, PTE, N/A, PTE</td>
</tr>
<tr>
<td>Non-occurrence</td>
<td>N</td>
<td>PTE</td>
<td>PTE, PTE, −</td>
</tr>
<tr>
<td>Aperiodic</td>
<td>A</td>
<td>PTE</td>
<td>PTE, PTE, N/A, PTE</td>
</tr>
</tbody>
</table>
| Cumulative aperi-
|odic             | A*     | N/A       | N/A, PTE, N/A, PTE |
| Periodic         | P      | N/A       | PTE, N/A, N/A, N/A, PTE |
| Termination of   | !      | N/A       | N/A, N/A, N/A, N/A, PTE |
| Logical event    | E[Q]   | E[Q]      | N/A, PTE, PTE, N/A, PTE |

<sup>a</sup>Temporal Calculi for Communicating Systems  
<sup>b</sup>Real-Time Logic  
<sup>c</sup>Chronicle Recognition  
<sup>d</sup>Interval-based Event Algebra by Carlson and Lisper [CL03]  
<sup>e</sup>Event Pattern Language [MZ97]

occur whenever a method is invoked; *temporal* events that occur when the specified time is reached; and *control-flow* events that occur whenever there is an important change of the control flow in the database such as start of transaction, commit transaction, end of transaction etc.

In contrast to active databases, to enable testing of real-time systems Schütz [Sch94b] claims that the following primitive event types are necessary: synchronization events, asynchronous interrupts, and access to time. To summarize, the available primitive event types depends on the purpose of event monitoring. In our work, primitive event types are not emphasized.

### 4.4 Semantics of Event Operators

Briefly, the semantics of an event operator is that if there are occurrences of the contributing event types that fulfill the constraints of the event operator, then there is an occurrence of the composite event type whose principal
4.4 Semantics of Event Operators

operator is this event operator. The semantics of these event operators can be defined using first-order predicate logic. The $\text{occ}(E, [t, t'])$ predicate introduced by Galton et al. [GA01] is used\(^2\). This predicate is true if an occurrence of $E$ that started at time $t$ and terminated at time $t'$ has occurred.

**Definition 4.2**

**Primitive event occurrence:**

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

\[\Box\]

Defs 4.3 to 4.11 on pp. 47–49 define how occurrences of the operands of an operator can contribute to a composite event occurrence [GA01].

**Definition 4.3**

**Disjunction:**

$\text{occ}(E_1 \lor E_2, [t, t']) \triangleq \text{occ}(E_1, [t, t]) \lor \text{occ}(E_2, [t, t'])$

\[\Box\]

**Definition 4.4**

**Sequence:**

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

\[\Box\]

**Definition 4.5**

**Conjunction:**

$\text{occ}(E_1 \land E_2, [t, t']) \triangleq$

\[
\exists t_1 \leq t'_1, t_2 \leq t'_2 (\text{occ}(E_1, [t_1, t'_1]) \land \text{occ}(E_2, [t_2, t'_2]) \land t = \min(t_1, t_2) \land t' = \max(t'_1, t'_2))
\]

\[\Box\]

\(^2\)In the paper by Galton et al., the $\text{occ}$ predicate is written $O$. Since this can be confused with the big-oh notation for algorithmic complexity, $\text{occ}$ has been chosen instead.
Def. 4.5 on the previous page is a rewritten version of Galton et al., since they do not employ the min and max functions. Galton et al.’s version considers all four combinations of occurrence possibilities of the contributing event occurrences.

**Definition 4.6**

Non-occurrence:

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

□

In other words, a non-occurrence of \(E_2\) occurs if no \(E_2\) event occurs in an interval opened by an \(E_1\) occurrence and closed by any \(E_3\) occurrence.

**Definition 4.7**

Periodic:

\[
\text{occ}(P(E_1, T, E_2), [t, t']) \triangleq \\
\exists i, t_1 \leq t_2 (i \in \mathbb{N} \land \text{occ}(E_1, [t_1, t_2]) \land t = t' = i \ast T + t_2 \land \\
\forall t_3 \leq t_4 (t_2 \leq t_3 \land t' \geq t_4 \Rightarrow \neg \text{occ}(E_2, [t_3, t_4]))
\]

□

In other words, the periodic event expression holds true for a multiple of \(T\) starting from the end of the time span of an \(E_1\) occurrence until an \(E_2\) occurrence. Basically, this event operator is used to generate periodic event occurrences.

**Definition 4.8**

Aperiodic:

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

□
In other words, an aperiodic event occurs if an $E_2$ event occurs in an interval defined by event occurrences of type $E_1$ and $E_3$. To be more precise, an aperiodic event occurs with the same time span as an $E_2$ occurrence that succeeds (or is met by) and $E_1$ occurrence and there are no $E_3$ occurrences preceding (or meeting) the $E_1$ occurrence that terminates before or at the same time as the $E_2$ occurrence.

**Definition 4.9**
Cumulative aperiodic expression:
\[
\text{occ}(A*(E_1, E_2, E_3), [t, t']) \triangleq \exists t_1 < t_2 (\text{occ}(E_1, [t, t_1]) \land \text{occ}(E_3, [t_2, t']))
\]

Def. 4.9 is only listed due to completeness, since we rejected the cumulative aperiodic event operator in the licentiate thesis [Mel98a], because it introduce unpredictable resource requirements.

Def. 4.9 is equivalent to $E_1;E_3$ with respect to the time span $[t, t']$. The difference between the sequence operator and the cumulative aperiodic operator is that an event occurrence of the sequence operator only consists of event occurrences of the contributing event types (in this case: $E_1$ and $E_3$), whereas an event occurrence of the cumulative aperiodic event operator consists of a set of $E_2$ occurrences between an $E_1$ occurrence and an $E_3$ occurrence. However, since we only care about the time span throughout which an event occurrence occurs, the $E_2$ is not considered in Def. 4.9.

**Definition 4.10**
Relative temporal:
\[
\text{occ}(E_1+T, [t, t']) \triangleq \exists t_1 (\text{occ}(E_1, [t, t_1]) \land t' = t + T \land t' \leq \text{now})
\]

The definition of Def. 4.10 is extended with now in this work.

**Definition 4.11**
Termination of:
\[
\text{occ}(E!, [t', t']) \triangleq \exists t (\text{occ}(E, [t, t']))
\]
This operator is introduced by Galton et al. [GA01] to allow specification of the old semantics that is only using the termination time of event occurrences. For example, $(E_1!);(E_2!)$ is treated as if the contributing event types are primitive even though they may be composite event types.

Additionally, we introduce the following definition:

**Definition 4.12**

Logical occurrence:

$$\text{occ}(E\{Q\}, [t, t']) \triangleq \text{occ}(E, [t, t']) \land Q$$

In other words, $E\{Q\}$ occurs if and only if the predicate $Q$ holds. Basically, this is a filter that allows us to select what events are propagated to contributed event types as well as subscribers.

### 4.4.1 Interval and Termination Semantics

In event detection, it is necessary to separate between interval and termination semantics of events. The definitions (Def:s 4.3 to 4.11 on pp. 47–49) are based on interval semantics, but the terminator operator (in Def. 4.11 on the previous page) allows event expressions to use termination semantics [GA01]. *Termination semantics* states that event occurrences are instantaneous and their time of occurrence is when event occurrences terminate, whereas the *interval semantics* allows non-instantaneous event occurrences and their time of occurrence is an interval.

Termination semantics was named detection semantics by Galton et al. [GA01]. However, this name indicates that the time delay between the termination of an event occurrence and the detection of the termination is instantaneous (or, at least, negligible [CM94]). In our work, this delay is not assumed to be negligible as in the model of Marinescu et al. [MLC90] and thus termination and detection are separated. Therefore, the term termination semantics is used instead of detection semantics.

All the languages addressed in this chapter follow termination semantics except the refinement of Solicitor [Mel98a] in our work, the interval-based event algebra by Carlson et al. [CL03], and the variant of Snoop by Adaikkalavan and Chakravarthy [AC02, Ada02]. According to Galton et al., one problem with termination semantics is that Eq. 4.1 on the facing page holds.
4.5 Event Contexts: Order, Reuse, and Time Constraints

∀t(occ( ((E₁!); ((E₂!);(E₃!)))!],[t, t]) ⇔
occ( ((E₂!);((E₁!);(E₃!)))!],[t, t]) (4.1)

In other words, given that all events only have termination semantics (emulated in the expression by using the terminator operator), then, for example, \( \text{occ}(E₁, [2, 2]) \), \( \text{occ}(E₂, [1, 1]) \), and \( \text{occ}(E₃, [3, 3]) \) holds true, it implies that both the predicate \( \text{occ}((E₁!; (E₂!; E₃!))!],[3, 3]) \) and the predicate \( \text{occ}((E₂!; (E₁!; E₃!))!],[3, 3]) \) are true. This unintuitive meaning of sequence has lead to a variation of Snoop [AC02, Ada02] that uses the interval semantics instead.

Note that it is technically possible to express interval semantics in event specification languages using termination semantics. However, the abstraction is lost since it is necessary to specify all event expressions in terms of primitive event types only. For example, assume that \( Eₐ = E₁; E₂ \), \( E₉ = E₃; E₄ \) and \( E₇ = Eₐ; E₉ \), then \( E₇ \) can be written as \( E₇ = ((E₁; E₂); E₃); E₄ \) by specifying \( E₇ \) in terms of primitive event types rather than composite event types.

4.5 Event Contexts: Order, Reuse, and Time Constraints

One issue addressed within Snoop (e.g., [CKAK94, CYY98]) are event contexts. An event context defines what combination of contributing event occurrences are meaningful to combine into a composite event occurrence in a specific situation. An event context constrains what occurrences are meaningful. The least constrained (or unconstrained) event context is named general context\(^3\) by Chakravarthy et al. [CM94]. The algorithmic time complexity of event composition in general event context is conjectured to be exponential in terms of generating all composite event occurrences with respect to the number of event operators in an event expression. This conjecture is based on that the general event context is similar to making the Cartesian product out of all incoming event occurrences, the same problem that Snodgrass [Sno88] faces (in Section 2.2 on p. 19). This is obviously not acceptable for complex event expressions. Therefore, an event context can be viewed as

\(^3\) Also known as unconstrained event context.
applying pragmatic constraints to event composition. To discuss event contexts, it is necessary to introduce further formal definitions. Additionally, it is necessary to distinguish between the initiator and terminator occurrences of a composite event occurrence.

4.5.1 Basic Formal Definitions for Event Contexts

The semantics defined in Def:s 4.3 to 4.11 on pp. 47–49 does not define the processing, a desirable feature for investigating properties of event composition. Even though it is possible to augment these definitions with sub-predicates addressing event contexts as, for example, by Adaikkalan and Chakravarthy [AC02, Ada02], processing is not defined. As is discussed in Ch. 6, the definitions also become less intuitive. Given all this, a different notation to express that an event has occurred is introduced to avoid confusion with first-order predicate logic that defines the semantics of event operators in a well-defined way.

Each occurrence of an event type is denoted by an event token that is defined as the relation (or constructor) $\Gamma(E, [t, t'])$; read “an event of type $E$ has occurred throughout the interval $[t, t']$”. N.B. the symbol $\Gamma$ is chosen since event occurrences are generated. In contrast to $\text{occ}(E, [t, t'])$, the $\Gamma(E, [t, t'])$ is not deduced by applying Def:s 4.3 to 4.11 on pp. 47–49 to asserted facts defining primitive event occurrences, but all $\Gamma(E, [t, t'])$ are asserted. Primitive event occurrences are asserted by the monitored object, but composite event occurrences are asserted by event composition processing. This event composition processing heeds the constraints introduced by event contexts.

The set of all syntactically well formed tokens are $T = \{\Gamma(E, [t, t'])\}$. This set only represents syntactically well formed tokens, since their membership does not mean that any of them represents an occurrence. Finally, the symbol $\gamma$ is also used to denote event tokens.

The functions applied to event occurrences are:

**Definition 4.13**

**Occurrence functions:**

Let $\gamma \overset{\lambda}{=} \Gamma(E, [t, t'])$ in

- $\text{type}(\gamma) = E$
- $\text{span}(\gamma) = [t, t']$
4.5 Event Contexts: Order, Reuse, and Time Constraints

Definition 4.14
Interval functions:
Let \([t, t']\) be an interval in
\[
\begin{align*}
\text{start}([t, t']) &= t \\
\text{end}([t, t']) &= t' \\
\|t, t'| &= t' - t
\end{align*}
\]

□

The \(G\) (generated set) represents the event history and contains all asserted primitive and composite event occurrences such that:

Axiom 4.1
Occurrence history:
\(G \subseteq T \land \forall \gamma \in G \ (\text{end(span(\gamma)))} \leq \text{now})
□

In other words, all members of \(G\) are well-defined tokens where the termination time of each occurrence represented by an event token in \(G\) is less than or equal to the current time.

N.B., since the emphasis is on processing, it is necessary to distinguish between the contents of \(G\) before and after event composition; therefore, \(G', G''\) etc. denotes the contents of \(G\) after consecutive processing steps, where \(G \subseteq G' \subseteq G'' \ldots\)

Finally, the \(\mathbb{OP}_R\) set contains the temporal ordering operators applicable to event occurrences. The actual content of this set depends on the event specification language. For example, in centralized systems it is sufficient that \(\mathbb{OP}_R = \{<\}\) since no two event occurrences can be simultaneous, whereas in distributed systems with termination semantics \(\mathbb{OP}_R = \{<, \leq, =, ||\}\) (where \(||\) means concurrent) can be sufficient. Another example is the difference between termination and interval semantics; Roncancio [Ron97] suggests that Allen’s temporal interval operator [All83] is a sound basis as temporal ordering operators for event composition.

Alternative Representation of Event Occurrences: In active database articles, the notation \(e_j^i\) is used to describe primitive event occurrences
(or named composite event occurrences) as tokens, where $t$ is the time of occurrence and $j$ is the index of the event type. This notation has two shortcomings. Firstly, it is difficult to use this format in predicates, since it requires knowledge about the structure of the representation of event occurrences that have been abstracted away in the relation $\Gamma(E,[t,t'])$. For example, assume that $G = \{\Gamma(E_1,[1,1]), \Gamma(E_2,[2,2]), \Gamma(E_3,[3,3])\}$ and that we are monitoring $E = E_1 \triangle E_2 \triangle E_3$, then we know that $\Gamma(E,[1,3])$ has occurred in contrast to the alternative representation stating that $e_1^1 \triangle e_2^2 \triangle e_3^3$ has occurred. To define the span function for $E$ we can write $\text{span}(\gamma) = [t, t']$ where $\gamma = e_1^1 \triangle e_2^2 \triangle e_3^3 \land t = \min(t_1, t_2, t_3) \land t' = \max(t_1, t_2, t_3)$. That is, to define functions based on event occurrences using this representation constrains the functions to unnecessary details. Secondly, the notation only implies a termination time, whereas an interval time is desirable to express. Even though this alternative notation may be viewed as more readable, it is only applicable to small examples, since the representation of event occurrences tends to become cluttered for larger event expressions (e.g., [Mel98a, p. 32]). Therefore, the choice has been to express the occurrence as a relation, since the syntactical construct of relations lends itself to formal definitions.

### 4.5.2 Initiator and Terminator Occurrences of Event Types

An important fact in the discussion of event contexts are initiator and terminator occurrences. Briefly, an initiator occurrence has initiated a composite event occurrence, whereas a terminator occurrence has terminated a composite event occurrence.

The event operators constrain the event types of initiators and terminators. For example, only an event occurrence of $E_1$ may be an initiator occurrence of $E_1 \triangle E_2$, whereas an occurrence of both $E_1$ and $E_2$ may the initiator of an $E_1 \triangle E_2$ event.

The difference between initiator and terminator occurrence of event types can be exemplified as follows; given the specification $E_1 \triangle E_2$ and $G = \{\Gamma(E_1,[1,1]), \Gamma(E_2,[2,2])\}$ results in $\Gamma(E_1 \triangle E_2,[1,2]) \in G'$ where $\Gamma(E_1,[1,1])$ is the initiator and $\Gamma(E_2,[2,2])$ is the terminator. Similarly, $G = \{\Gamma(E_2,[1,1]), \Gamma(E_1,[2,2])\}$ results in the $\Gamma(E_1 \triangle E_2,[1,2]) \in G'$ where $\Gamma(E_2,[1,1])$ is the initiator and $\Gamma(E_1,[2,2])$ is the terminator.

Zimmer et al. [ZU99] also define interiors. An interior event occurrence is enclosed in an interval such as $E = A*(E_1, E_2, E_3)$ where $E_2$ occurrences
4.5 Event Contexts: Order, Reuse, and Time Constraints

are interior occurrences of $E$ occurrences, $E_1$ are initiator occurrences, and $E_2$ are terminator occurrences. However, we are only interested in initiator and terminator occurrences, since we are not interested in cumulative event operators such as $A*(E_1, E_2, E_3)$ because they introduce unpredictable resource requirements; further, the other event operators, Defs 4.3 to 4.11 on pp. 47–49, do not include anything but an initiator and a terminator in the occurrence of the composite event type. For example, the initiator and terminator occurrences of $E = A(E_1, E_2, E_3)$ are both of $E_2$.

4.5.3 General Event Context

To further the understanding of the general context, the following example is introduced:

Example 4.1
Assume $\mathcal{G} = \{\Gamma(E_1, [1,1]), \Gamma(E_2, [2,2]), \Gamma(E_2, [3,3]), \Gamma(E_1, [4,4]), \Gamma(E_1, [5,5]), \Gamma(E_2, [6,6])\}$. If we monitor $E_{12} = E_1; E_2$, then the $occ$ predicate holds true for $\{\Gamma(E_{12}, [1,2]), \Gamma(E_{12}, [1,3]), \Gamma(E_{12}, [1,6]), \Gamma(E_{12}, [4,6]), \Gamma(E_{12}, [5,6])\}$. This is depicted in Fig. 4.1 on the following page.

4.5.4 The Semantics of Event Contexts

To further the understanding of the other event contexts, the following example is introduced:

Example 4.2
To give an example of an event context, consider a centralized computer system with one processor. Event occurrences are totally ordered, since no two event occurrences can be generated simultaneously. In this situation, it can be desirable to form composite events strictly based on the (historical) order of occurrence. This context is named *chronicle context* [CKAK94].

For example, if it is desirable to monitor file processing in order to validate generated or updated files when they are closed, then it is desirable to monitor open file events (named “ofc”) followed by close file completed events (named “cfc”). That is, $ofc; cfc$. Assume that each pair of application and file has a unique event type of both $ofc$ and $cfc$. Moreover, assume
that it is impossible to open an opened file and close a closed file. Without loss of generality, it is possible to use the chronicle context to constrain which event occurrences contribute to composite event occurrences since it is assumed that an application opens a file once and closes it once during processing.

Assume $G = \{ \Gamma(ofc, [1,1]), \Gamma(cfc, [2,2]), \Gamma(ofc, [5,5]), \Gamma(cfc, [9,9]) \}$. Then, $\Gamma(cfc; ofc, [1,2])$ and $\Gamma(cfc; ofc, [5,9])$ are meaningful events, whereas $\Gamma(cfc; ofc, [1,9])$ and $\Gamma(cfc; ofc, [2,9])$ are not in this situation. In chronicle context, once an event occurrence has contributed to a composite event occurrence it cannot contribute to another composite event occurrence of the same type (in this case $ofc; cfc$). However, contributing occurrences can be used to form composite occurrences of other event types (e.g., $ofc \triangle E_x$).

This context reduces the computational complexity, since it is not necessary to combine all possible initiators with all possible terminators meeting the operator constraint.
Chakravarthy et al. [CKAK94] claim that general event context is only meaningful as a reference since its (algorithmic time) complexity leads to non-scalable solutions. Then, they identify the following contexts:

**Recent context:**

The recent event context is actually referred to as “most recent event context” in, for example, the work by Chakravarthy et al. [CKAK94]. However, we have chosen to call it recent context, since this is the intuitive meaning [BMH99]. In this context, only the most recent occurrences are allowed to contribute to composite events. Typical application of this context is when monitored objects are monitored using oversampled sensors. This situation is common in real-time systems. Moreover, it can also be used to trigger checks of weak integrity constraints in databases that do not need to pinpoint what records or objects that have been updated. For example, assume that an integrity constraint is that the sum of all salaries may not be over 50 percent of the income of a company. Then, assume that a series of updates are made to the salaries within a transaction. It is not necessary to know which records have been updated to check the integrity constraint, since the check of the integrity constraint must summarize all records; it is only necessary to know that at least one update has been made to the salaries to check this integrity constraint.

Given Example 4.1 on p. 55, the event occurrences $\Gamma(E_{12}, [1,2]), \Gamma(E_{12}, [1,3]),$ and $\Gamma(E_{12}, [5,6])$ has occurred according to the recent context as depicted in Fig. 4.2 on the following page. The view of time (in Section 2.3 on p. 22) is right-branching. Each new terminating event occurrence brings a new decision point, in which the assumption is that a composite event is discovered with the most recent initiator.

**Chronicle context:**

In this context, only occurrences with matching order of occurrence are allowed to contribute to composite events. This context is useful in situations where the constraints on the behavior of the monitored object enable event composition to compose events based on order. Centralized computer systems with a single processor are a typical example where such situations can be found as well as distributed systems where order is maintained (e.g., MARS [KDK+89, KV94], Delta-4 XPA [Pow91, KV94], Isis [BJ87]).
Given Example 4.1 on p. 55, \{Γ(E_{12}, [1, 2]), Γ(E_{12}, [4, 6])\} has occurred as depicted in Fig. 4.3. In the chronicle context, the view of time is parallel since each new initiator occurrence gives rise to a separate line of reasoning (i.e., which event has occurred).

**Continuous context:**
In this context, a terminator occurrence can be used together with several initiator occurrences to contribute to several composite event occurrences (one for each initiator occurrence). This can be used to study trends. An example briefly discussed by Chakravarthy et al. [CKAK94] is stock market analysis.
Given Example 4.1 on p. 55, the event occurrences \( \Gamma(E_{12}, [1, 2]), \Gamma(E_{12}, [4, 6]), \) and \( \Gamma(E_{12}, [5, 6]) \) has occurred in the continuous event context. This fact is depicted in Fig. 4.4. In this context, the view of time is left-branching since each new terminator occurrence can result in several composite event occurrences relating to the same situation. In the example, both \( \Gamma(E_{12}, [4, 6]) \) and \( \Gamma(E_{12}, [5, 6]) \) represent two different (left) branches of the same reasoning (terminating at the same time).

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E_{12} )</td>
<td>( \Gamma(E_{12}, [1, 2]) )</td>
<td>( \Gamma(E_{12}, [4, 6]) )</td>
<td>( \Gamma(E_{12}, [5, 6]) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- \( \square \) Primitive event occurrence
- \( \square \square \) Composite event occurrence
- \( \times \) Invalidated event occurrence

Figure 4.4: Composition of \( E_{12} = E_1; E_2 \) in continuous event context

Cumulative context:
Similarly to continuous context, a terminator occurrence can be used with several initiator occurrences to contribute to, in contrast to the continuous context, one composite event occurrence.

Given Example 4.1 on p. 55, \( \{ \Gamma(E_{12}, [1, 2]), \Gamma(E_{12}, [4, 6]) \} \) has occurred as depicted in Fig. 4.5 on the following page. As in continuous context, the view of time is left-branching, since several initiator occurrences are combined with a terminator occurrence. The difference is that in cumulative context, the result is aggregated into one composite event occurrence for each terminator occurrence rather than one composite event occurrence for each terminator occurrence. For example, both branches of reasoning represented as two separate event occurrences in the continuous event context \( \{ \Gamma(E_{12}, [4, 6]), \Gamma(E_{12}, [5, 6]) \} \) are represented by one event occurrence \( \{ \Gamma(E_{12}, [4, 6]) \} \) where \( \Gamma(E_1, [4, 4]), \Gamma(E_1, [5, 5]), \) and \( \Gamma(E_2, [6, 6]) \) are contributors.
Out of these, Buchmann [Buc94] state that support for recent and chronicle event contexts is a minimum requirement for real-time systems without significant substance such as an argument based on the time complexity of event composition. Moreover, he disallows composite event types for critical tasks without significant substance; composite event types are dismissed since the event composition is considered to be too inefficient. In contrast to Buchmann [Buc94], Milne et al. [MNG+94] use chronicle recognition for event monitoring in real-time systems and Chronicle recognition follows the general event context. In this Ph.D. thesis, the algorithmic time complexity of contexts is pursued further to shed light on what contexts that can be used for real-time systems.

To exemplify the differences between the event contexts, consider the resulting event occurrences in Table 4.4 for the event type $E_{12}$. The general event context is omitted since it does not introduce any constraints that need to be explained in further detail. There are differences in reuse of contributing event occurrences and order of occurrence. In the recent and continuous contexts, a contributing event occurrence may be reused; whereas
in chronicle and cumulative contexts, a contributing event occurrence cannot be reused. In the latter contexts, contributing initiator event occurrences are consumed when they form composite event occurrences. In the chronicle event context, terminator event occurrences are also consumed when they form a composite event occurrence. Concerning order, the chronicle context follow a strict historical order of event occurrences, whereas the other contexts consider the order of event occurrences relative to the terminating event occurrence. In recent context, the most recent initiator occurrence is used. In chronicle and cumulative contexts, the most recent initiators that have not been used to form composite event occurrences are used.

An additional property of event contexts is the implicit invalidation of event occurrences due to that they are not selectable; an invalidated event occurrence may not contribute to the occurrence of future composite event occurrences of the event type in question. However, an event occurrence that has been invalidated with respect to a specific event type is not invalidated with respect to other event types that it can contribute to. For example, in Table 4.4 on the facing page the $\Gamma(E_2, [4,4])$ is invalidated for $E_{12}$ in the continuous event context since it cannot be used to form an $E_{12}$ event occurrence.

To summarize, event occurrences may either be used event occurrences (i.e., contributes to a composite event) or invalidated event occurrences (i.e., cannot contribute to composite event occurrences in the future). In both cases, they are used or invalidated with respect to the event types they contribute to. In some contexts, the use of an event occurrence results in consumption of this occurrence. The contexts also implies a constraint on the order of event occurrences that may be absolute to all other event occurrences or relative to a terminator. The aforementioned event contexts define meaningful combinations of constraints for specific situations.

### 4.5.5 Overlapping and Non-overlapping Contexts

Additionally, consecutive composite event occurrences formed by the same event operator in an event context may be overlapping or non-overlapping. By default, event contexts allow consecutive overlapping event occurrences as can be seen in the recent and continuous contexts in Table 4.4 on the facing page (e.g., $\Gamma(E_{12}, [2,3])$ and $\Gamma(E_{12}, [2,4])$ are two consecutively overlapping event occurrences formed by the same event operator). As a counter example, assume that the event contexts are non-overlapping; then recent
context results only in $\Gamma(E_{12}, [2,3])$, and, in a similar way, continuous context results only in $\Gamma(E_{12}, [1,3])$. A remark is that with respect to the terminator occurrence, the non-overlapping recent context uses the combinations with the shortest duration to form composite event occurrence. In contrast, the non-overlapping continuous context uses the combinations (of unused contributing occurrences) with the longest duration to form composite event occurrences.

### 4.5.6 Time Constraints

If we are monitoring outside a temporal scope such as a transaction, then it is necessary to use validity intervals as discussed in Section 3.5 on p. 35. In addition to the reasons mentioned in that section, there is a problem that if there are initiators occurring but not terminators, then the number of initiator occurrences that must be managed can grow indefinitely.

Validity intervals are addressed in Chronicle recognition [Dou96] where this validity interval is named “Chronicle deadline”. A completed “Chronicle” can be viewed as a composite event occurrence. An initiated Chronicle that passes the deadline is assumed to be impossible to complete and, therefore, the initiated Chronicle is discarded.

Briefly, a Chronicle can be expressed as a disjunction of sequences of event types. It can be viewed as if the terminator operator (in Def. 4.11 on p. 49) is applied to each event type and subexpression. Additionally, there are time constraints between pairs of event occurrences that can be expressed as event parameter conditions. The event composition of Chronicle recognition follows a general context; to be more precise the disjunctions follow a non-overlapping general context and the sequences follow an overlapping general context. It is possible to let the disjunctions also follow a overlapping general event context, however, this is considered inefficient by Dousson et al. [DGG93, Dou96].

For example, assume that $G = \{\Gamma(E_1, [1,1]), \Gamma(E_2, [3,3]), \Gamma(E_2, [12,12]), \Gamma(E_3, [20,20])\}$ and that $E_c = E_a; E_b[\text{start}(\text{span}(E_a)) - \text{end}(\text{span}(E_b)) < 10]$ (where $E_a = E_1!$ and $E_b = (E_2; E_3)!$) is monitored in general event context. In other words, the temporal distance between an $E_1$ occurrence and an $E_3$ occurrence may be at most 10 time units. The result of this is that $\Gamma(E_c, [1,20])$ has occurred where $\Gamma(E_1, [1,1]), \Gamma(E_2, [12,12]),$ and $\Gamma(E_3, [20,20])$ are contributors. The $\Gamma(E_2, [3,3])$ is discarded since it does not meet the time constraint on the event expression. However, it is not discarded until
4.6 Managing Concurrent Event Occurrences

Now $>$ 13. Therefore, during time 12 to 13, the event composer has two possibilities to complete composing event occurrences of $E_c$.

This feature of Chronicle recognition is called duplication by Dousson et al. [DGG93, Dou96], since it is necessary to keep each possibility in separate but related instances of Chronicle models. One important aspect of Chronicle recognition is when a solution is found, all related duplicates can be discarded. Therefore, the disjunctions can be viewed as if they follow the non-overlapping general event context. In contrast, all subexpressions based on sequences follow an (overlapping) general event context.

Assume that $\Gamma(E_3, [21, 21]) \in G$ too, then this still results in that only $\Gamma(E_c, [1, 20])$ has occurred since the duplicate $\Gamma(E_c, [1, 21])$ is overlapped by the former.

Validity intervals have been addressed indirectly in real-time logic (e.g., [CJD91, JRR94]), since a basic expression in real-time logic is a relation concerning the temporal distance between two event occurrences.

In both approaches, it has been recognized that it is necessary to pre-compile time constraints into an internal representation. The reason is that time constraints may be contradictory, redundant, or there may be an earlier point in time that a time constraint can be checked than what has been specified. This latter issue is important for early detection of satisfaction or violation of time constraints (e.g., Liu et al. [LMK98]).

4.5.7 Combining Constraints

It is desirable to combine the different constraints on event composition, since it reduce the algorithmic time complexity of event composition algorithms. So far, monitoring of real-time logic is the only language that allows this; however, only a restricted set of combinations of constraints is allowed.

4.6 Managing Concurrent Event Occurrences

Since order is an important issue for event contexts, concurrent event occurrences must be managed in distributed systems. Concurrent event occurrences are event occurrences that cannot be discriminated using their timestamps regardless of interval or termination semantics. Concurrent event occurrences can be managed by:

1. avoiding them, for example, monitoring in a centralized system or letting a central agent such as the leader in semi-active replication [KV94] arbitrarily determine the order of concurrent event occurrences
2. preventing them, for example, by introducing an artificial scheme that deterministically introduces an order among concurrent event occurrences

3. allowing them

The 2nd and 3rd alternatives are desirable to pursue, since any solution covers the special case (alternative) 1. Moreover, alternative 1 is not applicable to a large range of real-time systems, since many of them are distributed in such a way that a centralized agent is infeasible. In this Ph.D. thesis, alternative 2 is pursued, but the intention is to allow future extensions that cover alternative 3.

Alternative 2 has been employed in ensuring replicated services in distributed systems [Sch94a], and ensuring causal delivery of messages in real-time communication [Ver94b]. The basis is that messages (named requests by Schneider [Sch94a]) are not delivered to the recipients, even though they may be received earlier, until it is guaranteed that they are stable [Sch94a]. Messages are stable when there is no message that can precede them (in a distributed system). When a message is stable, then the message can be delivered to the recipients. In real-time communication, an additional requirement is that a replicated (broadcasted) message should be delivered at the same time to different recipients.

Alternative 3, has been considered by Schwidersky et al. [SHM95] and Liebig et al. [LCB99]. In both cases, no real-time communication is assumed. The solution by Schwidersky et al., is based on refining the timestamps of event occurrences (later corrected by Yang et al. [Yan99, YC99]). However, their algorithms for updating timestamps and comparing timestamps are computationally expensive. Two different protocols for distributed event monitoring were evaluated: a synchronous protocol and an asynchronous protocol. Since they had no real-time communication it was concluded that the synchronous protocols guaranteed delivery, but it achieves worse performance than the asynchronous protocol. However, in a distributed real-time system, the asynchronous protocol can also have the same guarantees as the synchronous protocol. Liebig et al. address open distributed systems where it is difficult to maintain $\delta_t$-precedence, by giving each timestamp an accuracy interval. Similarly to $\delta_t$-precedence, it is only a partial ordering (in Section 3.2.2 on p. 32). They propose an architecture based on a heartbeat algorithm to maintain stable event occurrences. The solution proposed by Liebig et al. is more efficient than those proposed by Schwidersky et al.
4.7 Formalization of Event Composition: Existing Approaches

An important issue is how event composition processing has been formalized for the different event specification languages, since a formalized model of processing is desirable for proving properties of event composition (e.g., resource-predictability) as well as investigating properties such as algorithmic complexity. There are two major issues concerning formalization: the outcome of event composition processing and the event composition processing itself. Many formalizations only consider the first issue (e.g., Snoop [CM94, CYY98, GA01, Ada02, AC02], ODE [GJS93], Interval-based event algebra for restricted event detection [CL03]), whereas other formalizations emphasize the latter issue (e.g., SAMOS [Gat94], TCCS [BG97], monitoring of RTL expression [LM99, LMK98, ML97a, ML97b, CJD91], Chronicle recognition [DGG93, Dou96], EPL [MZ97]. A common theme in the approaches formalizing the outcome is to start from a semantics such as in Def:s 4.3 to 4.11 on pp. 47–49 and define the result in general context (unconstrained) and then restrict this by applying a filter. For example, Chakravarthy et al. [CYY98], Adaikkalavan et al. [Ada02, AC02], Carlson et al. [CL03], and Jaeger [Jae97] follow this common theme. However, the actual processing to obtain the result can only be indirectly investigated.

<table>
<thead>
<tr>
<th>Event specification language/Prototype</th>
<th>Event contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chronicle recognition</td>
<td>general</td>
</tr>
<tr>
<td>EPL</td>
<td>all except cumulative</td>
</tr>
<tr>
<td>ODE</td>
<td>not applicable</td>
</tr>
<tr>
<td>RTL</td>
<td>recent, chronicle</td>
</tr>
<tr>
<td>SAMOS</td>
<td>chronicle</td>
</tr>
<tr>
<td>TCCS</td>
<td>chronicle</td>
</tr>
</tbody>
</table>

Table 4.5: Applicable event contexts in available formalizations

The current formalizations of processing are either locked to specific event contexts or they do not consider processing outside the scope of transactions. The current event contexts of the formalizations are summarized in Table 4.5. The only formal approach that is applicable to a majority of event contexts is EPL [MZ97], whereas the other formalizations are con-
strained to a subset of existing event contexts. For example, in monitoring of RTL expressions, the algorithms cannot be easily extended to manage other event contexts, since monitoring of RTL expressions disallows the universal quantifier [CJD91]; the universal quantifier is necessary for, for example, continuous event context where several initiator occurrences can be combined with one terminator occurrence.

EPL offers a machine-executable formalism (based on condition-action rules) that does not immediately lend itself to proofs. Event specifications are translated into Horn-clauses executed by a Prolog-like inference engine (DataLog$\_S$). Essentially, the Horn clauses need to be rewritten using quantifiers to allow proofs. The major problem is that the similarities and differences between event contexts are not immediately visible in their formalism. This is an issue that we remedy in our work.

These formalisms will be defined in more detail in Section 4.8.2 on the facing page and Ch. 6.

### 4.8 Evaluation of Event Specification Languages

In this section, the event specification languages are evaluated with respect to the syntax and semantics of the abstract syntax as well as the event contexts. Since the event contexts have an impact on the event operators and the languages only support the operators in specific event contexts, this section begins with an evaluation of event contexts and time constraints followed by the event operators.

#### 4.8.1 Support for Event Contexts and Time Constraints

The addressed event specification languages support different kind of constraints in terms of event contexts and support for time constraints. In this section, all possible combinations of overlapping and non-overlapping contexts with and without time constraints are considered. In Table 4.6 on the facing page, the result of this evaluation is described.

As can be seen, there are many combinations that are not supported by any language at all. Some combinations are not desirable since their algorithmic time complexity is high. For example, the general context without time constraints can lead to grave performance degradations in event composition processing. N.B. ODE does not have any support for event contexts, since the selection semantics is embedded in their event operators [ZU99].
Table 4.6: The support of event specification languages for event contexts and time constraints

<table>
<thead>
<tr>
<th>Overlapping or Non-overlapping</th>
<th>Time constraints</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlapping</td>
<td>No</td>
<td>Results in Table 4.7 on the following page</td>
</tr>
<tr>
<td>Overlapping</td>
<td>Yes</td>
<td>Only real-time logic has support for constraining event contexts: most recent and chronicle. Chronicle recognition uses general context for subexpressions.</td>
</tr>
<tr>
<td>Non-overlapping</td>
<td>No</td>
<td>Only Snoop has support for the event contexts. General context is not supported in practice.</td>
</tr>
<tr>
<td>Non-overlapping</td>
<td>Yes</td>
<td>Only Chronicle recognition has support for contexts: general context</td>
</tr>
</tbody>
</table>

Further, event contexts is a static combination of constraints of order and use. Monitoring of real-time logic offers a more flexible scheme in terms of expressibility, in which it is possible to state constraints on recurring events. For example, it is possible to state the minimum interarrival time between two consecutive event occurrences of the same event type.

It is desirable to investigate the implications of the different variations of event contexts within one event specification language, since there is no complete investigation of them. For example, what impact does non-overlapping policy have on the time complexity of event monitoring?

4.8.2 Evaluation of Existing Event Operators

In Table 4.2 to 4.3 on pp. 45–46, some operators in the abstract syntax are expressed as “PTE” implying that it is possible to express the operator using the existing operators in the language. All languages use the termination time semantics with the inherent problem addressed by Galton et al. [GA01] (described in Section 4.4.1 on p. 50).
Event Composition: Syntax, Semantics, and Processing

<table>
<thead>
<tr>
<th>Event context</th>
<th>Solicitor</th>
<th>Snoop</th>
<th>REACH</th>
<th>SAMOS</th>
<th>ODE</th>
<th>TCCS</th>
<th>RTL</th>
<th>CR</th>
<th>CL</th>
<th>EPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>C</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td>Recent</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>C</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td>Chronicle</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>C</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td>Continuous</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>C</td>
<td>Y</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 4.7: The support of event specification languages for overlapping event contexts without time constraints

2. Interval-based event algebra for restricted event detection [CL03].
3. Carlson et al. [CL03] presents an event context that is dependent on the event operator. For all event operators except sequence the event context is similar to recent, whereas there is a more strict variant of context for the sequence event operator; they present this event context since it preserves algebraic properties of the event operators.

HiPAC, Snoop, SAMOS, ODE

Snoop [CM94], SAMOS [GD92], REACH [Buc94], and ODE [GJ92b] all support disjunction, conjunction, sequence, and various interval operators. An interval operator is an operator defined over a closed interval of an event history with either an initiator event type and a terminator event type defining the boundaries or, as in SAMOS [Gat94], a time interval. There are distinctions concerning interval operators that are discussed below.

The disjunction operator stems from HiPAC [DBB+88] and is exclusive because concurrent events are not allowed (cf. Section 4.6 on p. 63). That is, only one of the events in the disjunction can occur at the same time. In global event detection, for example, the X2S by Liebig et al. [LCB99] and the work by Schwidersky et al. [SHM95], the disjunctive operator is inclusive.

In SAMOS [Gat94, GD93], any event expression can be constrained by an in [t, t′] specification. For example, E1:E2 in [t, t′] meaning that an E1:E2 only occurs if it occurs in the half-open interval of between t and t′. This is similar to writing A(E3, E1:E2, E4) where t = end(span(Γ(E3, [t3, t])))) and t′ = end(span(Γ(E4, [t4, t′]))). Therefore, it is possible to specify an expression similar to the aperiodic expression in SAMOS. Moreover, since SAMOS includes a closure operator ‘∗’ (stemming from HiPAC [DBB+88]), it is possible to express an expression similar to the cumulative aperiodic expression...
4.8 Evaluation of Event Specification Languages

A*(E₁, E₂, E₃). In addition, SAMOS has a short form for writing a sequence of an event type itself named TIMES(n, E) implying that it occurs when n occurrences of E has occurred. For example, TIMES(3, E) means that three occurrences of the event type E results in the composite event occurrence TIMES(3, E).

The REACH prototype supports closure, history and not operator. The Ode prototype supports an event specification language based on regular expressions in which it is possible to define new event operators based on a set of basic operators [GJ92a]. In the comparison by Motakis et al. [MZ97] ODE is similar to EPL and Snoop event operators can be expressed in terms of EPL; thus, it is conjectured that the event operators in the abstract syntax can be defined in terms of ODE as well.

Temporal operators generate event occurrences based on a temporal specification. These specifications can be absolute, of which the P and P* operator in Snoop are examples, or relative to an event occurrence. In Snoop and REACH the relative temporal operator is as follows: E + T meaning that when an event e occurs E + T occurs at the time of occurrence of e plus T.

Temporal Calculi for Communication Systems

Temporal Calculi for Communication Systems (TCCS) is meant for specifying concurrent communicating tasks (agents) [Wan90]. Each expression is a definition of a task that communicates with other tasks using synchronization points similar to Communicating Sequential Processes. Bækgaard et al. [BG97] based their event control language on TCCS and introduced multiway synchronization between simple events (primitive events with one parameter only) and composite events.

An interesting part of TCCS specification is parameterized event types. Given Example 4.2 on p. 56, then it is possible to specify that the composite event should have equivalent parameters. For example, let the event specification ofc(aid: Application, fid: File) mean that an open file completed event has occurred within the application with identifier aid on file with the file identifier fid. This event type is parameterized indicating each ofc and combination of application and file is a unique event type.

They specify correct behaviors as never ending processes in CCS-style [Mil89]. To compose a sequence of ofc and cfc event occurrences in the same application operating on the same file is specified as the expression
Event Composition: Syntax, Semantics, and Processing

\[ X \equiv \text{ofc}(\text{aid}, \text{fid}).cfc(\text{aid}, \text{fid}).\text{ofccfc}(\text{aid}, \text{fid}).X \] where “?” denotes request, “!” denotes event enabling, “.” denotes sequence, “+” denotes disjunction and “≡” denotes “defined as”. In other words, the process \( X \) is forever waiting for \( \text{ofc} \) event occurrences followed by \( \text{cfc} \) event occurrences with equivalent \( \text{aid} \) and \( \text{fid} \) parameters. When this sequence occurs, then generate (or enable) the \( \text{ofccfc}(\text{aid}, \text{fid}) \) event occurrence and let process \( X \) return to the beginning of the specification (awaiting a sequence).

It is possible to specify the missing operators in TCCS. For example, to specify a non-occurrence of a write file completed event (\( \text{wfc} \)) in between an \( \text{ofc} \) and a \( \text{cfc} \) event occurrence is written as:

\[ \text{NO} \equiv \text{ofc}(\text{aid}, \text{fid}).\text{acfc}(\text{aid}, \text{fid}).\text{nowfc}(\text{aid}, \text{fid}).\text{NO} + \text{ofc}(\text{aid}, \text{fid}).\text{wfc}(\text{aid}, \text{fid}).\text{NO} \]

Each of the operators marked as possible to express can be expressed in TCCS, since it, in effect, is a specification for concurrent programs.

It is possible to express different contexts in TCCS, but it requires a low-level specification. For example, \( E = E_1;\langle\text{context: chronic}\rangle E_2 \) is the same as \( EP \equiv E_1.E_2.E!.EP \) in TCCS, whereas \( E = E_1;\langle\text{context: recent}\rangle E_2 \) is \( EP \equiv E_1.EP_2, EP_2 \equiv E_2.E!.EP_2 + E_1.EP_2 \). Note that parameters have been left out of this example. Some contexts such as general require an infinite set of TCCS equations.

Monitoring of Real-Time Logic

A unified approach between event composition and monitoring using real-time logic has been proposed by Liu et al. [LMK98]. They use a specification where the function \( @(E, i) \) means the time of occurrence of the \( i \)th event of type \( E \). Then, they express the monitored expression in a disjunctive normal form where the \( i \)th event occurrence of an event type relates to a delay to the \( i \)th event occurrence of an event type. For example, \( \exists i ((E_1, i)+T < @(E_2, i)) \) means that the minimum delay between the \( i \)th occurrence of \( E_1 \) and the \( i \)th occurrence of \( E_2 \) is \( T \).

They have two predicates \( \text{satisfy}(R_E) \) (and \( \text{fail}(R_E) \)) where \( R_E \) is a real-time logic expression. The predicate \( \text{satisfy}(R_E) \) is true if \( R_E \) holds, otherwise it is false (and vice versa for \( \text{fail}(R_E) \)). Additionally, they define

\[ e \]
4.8 Evaluation of Event Specification Languages

the $\text{occ}(E, i)^5$ predicate that is true if event type $E$ has occurred at index $i$. If $T = 0$, then according to Liu et al. [LMK98]6

$$\text{occ}((E_1!), \langle\text{context: chronicle}\rangle(E_2!), [t, t']) \triangleq \exists i (\text{satisfy}((\text{occ}(E_1, i) \land \text{occ}(E_2, i) \land @E_1, i < @E_2, i), t''') \land t = @E_1, i \land t' = @E_2, i \land t' \leq t''')$$

Since universal quantification is disallowed in monitored RTL expressions, then the existential quantifier of $i$ is left out and $\exists i (@E_1, i + T < @E_2, i)$ is written as $@E_1, i + T < @E_2, i$. It is also possible to express a recent context by using a negative index where $-1$ means the most recent context. This expression allows a flexible scheme for addressing different occurrences. For example, we can specify a minimum interarrival time between events of the same occurrence: $@E_1, i-1 + T < @E_2, i$.

The proposed unified approach is able to translate the semantics of the operators in the abstract syntax in a low-level fashion. For example, $((E_1!) \triangle (E_2!))$ can be expressed as $@E_1, i \leq @E_2, i \lor @E_1, i \geq @E_2, i$. In other words, the expressions using the operators in the abstract syntax can be translated into real-time logic expressions.

However, some of these translations can be computationally expensive. For example, Liu et al. [LMK98] defines

$$\text{occ}(A((E_1!), (E_2!), (E_3!)), [t, t']) \triangleq \text{satisfy}(\text{occ}(E_2, i) \land @E_3, @E_2, i) > @E_2, i, t''') \land t = @E_2, i \land t' = @E_2, i \land t' \leq t'''$$

where the $@E, t$ locates the closest index of $E$ at time $t$. With time indexing, this reversal function can be efficient at the cost of maintaining the time index. Otherwise, in the worst case, all event occurrences of $E$ must be processed to locate the index. Also, the $\text{occ}$ predicate is defined by the $\#$ operation.

Moreover, the event contexts that can be used in real-time logic are recent and chronicle event context. The other combinations of event operators and event contexts may be possible to translate into real-time logic. However, there is no proof of this.

---

5Liu et al. [LMK98] name it $\text{occ}$.  
6Since the $\text{satisfy}$ and $\text{occ}$ predicates have a different view of time, the definition has been refined to cater for this. However, the real-time logic expression is a direct quote from Liu et al. [LMK98].
Chronicle Recognition

As mentioned earlier, a Chronicle specification can be viewed as a disjunction of sequences where the terminator operator is applied to each event type. For example, let $E'_{23} = (E_2!);(E_3!)$, $E' = (E_1!);(E'_{23})!$, $E_{23}'' = ((E_3!);(E_2!)!)$, and $E'' = (E_2!);E''_{23}$ in $E = E''E''$; using the syntax by Dousson et al. [DGG93, Dou96], $E$ can be specified as

```
chronicle E
  E1 < E2 < E3
  E1 < E3 < E2
end
```

The time constraints are specified pairs of event types and an interval specification: $(E_2 - E_3)$ in $[10, 20]$. This means that the duration between the termination of $E_2$ and $E_3$ should be between 10 and 20. Let $E'_{23c} = E_{23}[10 \leq |\text{start(span}(E'_{23})| - |\text{end(span}(E'_{23})| | \leq 20]$, $E'_c = (E_1!);(E'_{23c})!$, $E''_{23c} = E''_{23}[20 \leq |\text{start(span}(E''_{23c})| - |\text{end(span}(E''_{23c})| | \leq 30]$, and $E''_c = (E_1!);(E''_{23c})!$ and $E_c = (E'_c)!\lor(E''_c)!$, then $E_c$ can be expressed as:

```
chronicle E_c
  E1 < E2 < E3
  E1 < E3 < E2
  (E2 - E3) in [20, 30]
  (E3 - E2) in [10, 20]
end
```

The composition in Chronicle recognition follows a time-constrained general context, where simple temporal networks are used to find a solution that meets the time constraints.

To see the difference in terms of contexts, consider the following for $E_c$: $G = \{\Gamma(E_1, [1,1]), \Gamma(E_2, [4,4]), \Gamma(E_2, [10,10]), \Gamma(E_3, [24,24])\}$. In this situation, $\Gamma(E'_{23}, [4,24])$ and $\Gamma(E'_{23c}, [10,24])$ are generated, which leads to that $\Gamma(E''_{23c}, [4,24])$ and $\Gamma(E'_{23c}, [10,24])$ are generated since both have a span matching the time constraint. In turn, these event occurrences lead to that $\Gamma(E'_c, [1,24])$ and $\Gamma(E'_c, [1,24])$ are generated and, finally, from these the $\Gamma(E_c, [1,24])$ and $\Gamma(E_c, [1,24])$ can be derived. To optimize Chronicle recognition, Dousson et al. [DGG93, Dou96] suggest that only one solution is presented to the subscriber of the Chronicle. Given that event occurrences
are processed in the order of termination time, then only \( \Gamma(E_c, [1,24]) \) based on \( \Gamma(E'_{23c}, [4,24]) \) is the result. This is the same result as non-overlapping general event context, since it does not allow resulting composite event occurrence with overlapping intervals.

A Chronicle specification can contain \( \text{holds}(Q) \) statements. If \( Q \) ever becomes false when a Chronicle has been initiated, then discard this Chronicle completely. The difference between \( E[Q] \) and \( \text{holds}(Q) \) is that \( E[Q] \) is evaluated when \( E \) has occurred, whereas \( \text{holds}(Q) \) terminates the current Chronicle. The \( \text{holds}(Q) \) can be partly expressed using the non-occurrence expression given that there are primitive events \( \text{notQholds} \) signalled when \( Q \) becomes false. For example, \( N(E_1, \text{notQholds}, E_2) \) terminates the composition of \( E_1;E_2 \) if \( \text{notQholds} \) occurs. The full expression is \( A(Q\text{holds}, N(E_1, \text{notQholds}, E_2), \text{notQholds}) \) meaning that this sequence is only signalled if \( Q \) holds for the duration of the composition for the non-overlapping general event context.

Additionally, a Chronicle specification contains event definitions as well as inclusion of other Chronicle specification. This inclusion allows us to base Chronicles on other Chronicles, a feature that is similar to composite events. However, since they use termination time semantics, this can lead to the problem addressed by Galton et al. [GA01].

**Interval-Based Event Algebra for Restricted Event Detection**

Carlson and Lisper [CL03] introduce an event algebra with an event context they name "restriction" where they maintain algebraic properties of the event specification language. First of all, they only support sequence, conjunction, disjunction, and non-occurrence; the non-occurrence is more general than the \( N(E_1, E_2, E_3) \), since it can take any \( E \) and state that it occurs if and only if not an \( E' \) has occurred during \( E \) denoted \( E - E' \). Secondly, their event context is similar to the recent context (cf. Def. 6.47 on p. 146) for all event operators but the sequence event operator; for the sequence event operator, a more restrictive form is provided.

**Event Pattern Language**

The Event Pattern Language (EPL) by Motakis and Zaniolo [MZ97] uses the Prolog-like DataLog language as the mechanism for event composition. They have an extensive way of defining logical event types out of primitive event types (called qualified basic event types). To each event type
E, a satisfaction predicate sat_E is derived: if \( E = \text{eventkind}_R(R(X), q(X)) \) then \( sat_E(X) \leftarrow \text{eventkind}_R(R(X), J, q(X)) \). where \text{eventkind} is \text{ins} for insert, \text{del} for delete, and \text{upd} for update; R is a table, and X are the columns of the table R. For example, assume that the relation \( ACC = (\text{Accno}, \text{Owner}, \text{Type}, \text{Time}) \) and we want to define an event type that occurs if \( \text{Type} = \text{"Savings"} \), then we would state \( E = \text{ins}(ACC(X), X.\text{Type} = \text{"Savings"}) \); this would be translated into [MZ97]:

\[
\text{ins}_\text{ACC}(\text{Accno}, \text{Owner}, \text{Type}, \text{Time}, J) \leftarrow \\
\text{hist}_\text{monit}(\text{ins}, \text{"ACC"}, \text{Time}, J), \\
\text{insertedACC}(\text{Accno}, \text{Owner}, \text{Type}, \text{Time}). \\
\text{sat}_E(\text{Accno}, \text{Owner}, \text{Type}, \text{Time}) \leftarrow \\
\text{ins}_\text{ACC}(\text{Accno}, \text{Owner}, \text{Type}, \text{Time}, J), \\
\text{Type} = \text{"Savings"}
\]

where \( J \) is the order in which something occurred (introduced to distinguish concurrent event occurrences), \( s(J) \) is a successor of \( J \) (i.e., \( \forall J (J \in \mathbb{N} \Rightarrow J < s(J)) \)) and \text{hist}_\text{monit} \text{is defined as:}

\[
\text{hist}_\text{monit}(\text{nil}, \text{nil}, 0000, 0)). \\
\text{hist}_\text{monit}(E, R, T2, s(J)) \leftarrow \\
\text{hist}_\text{monit}(\_\_T1, J), \\
\text{hist}(E, R, T2), \\
\text{evt}_\text{monit}(E, R), \\
\neg\text{between}(T1, T2). \\
\text{between}(T1, T2) \leftarrow \\
\text{hist}(E, R, T), \\
\text{evt}_\text{monit}(E, R), \\
T1<T, T<T2.
\]

where \text{hist} is a global history of event occurrences, and \text{evt}_\text{monit} is a record of primitive event types for a specific table.

In other words, the predicate \text{hist}_\text{monit} \text{is true for an event occurrence that is successive to } J, \text{that has terminated at time } T2 \text{ and is of a primitive event type, and there are no event occurrences between the event occurrence at stage } J \text{ and stage } s(J). \text{The first event occurrence is a dummy} \text{for event type } \text{nil} \text{ and relation } \text{nil}. \text{The } sat_E \text{ holds true for an account, an owner, a type, a balance that has been inserted at time } Time \text{ where } \text{Type} = \text{"Savings"}. \text{The } \text{ins}_\text{ACC} \text{ predicate holds true for an account, owner, type, balance at}
4.9 Architectural Issues of Event Detection

Event composition is realized as an algorithm employing some mechanism operating on some data structure that contains the filtered event log. Usually, this algorithm is split into two parts: (i) one generic part that controls what event type that should be evaluated and (ii) a specific procedure for each event type (a combination of principal event operator of the event type, and the event context of the event type) that composes event occurrences out of the contributing event types. For example, in Snoop [CM94, CKAK94] the generic part is the (remote) produce call semantics providing implicit scheduling of evaluation and the specific part is a procedure call (actually a method call) to an algorithm representing a combination of event operator semantics as well as event context semantics. Another example is SAMOS [Gat94], where the generic part is the Petri-net execution engine and the specific part are Petri-net patterns representing event types that can be combined into composite event types.

The filtered event log is often partitioned into data structures that contain only the contributing event occurrences relevant to a specific event type. For example, for $E = E_1; E_2$ event type $E$ only need to keep track of event occurrences of event types $E_1$ and $E_2$; in, for example, Snoop [CM94, CKAK94]
the contributing event occurrences of a contributing event type are placed in a queue between the contributing event type and the contributed event type; the procedures in Snoop representing a combination of event operator semantics and event context semantics process these queues. In our work, an abstraction of these data structures called connector sets are introduced in Section 7.1 on p. 152.

It is desirable to enable aggregation of the evaluation of a set of event types into a task, for example, the data structures of an event type can be aggregated into the same task that executes the procedures for event composition of that event type. This enables better control over the timeliness of a real-time system, since it allows a real-time scheduler to control the execution of evaluation. Further, it allows distribution of event monitoring as well as replication of event monitoring for fault-tolerance. Given tasks, results of event composition are distributed either via intra-task or inter-task communication.

Event composition is based on the following services where service S2 to S5 were addressed by Mellin et al. [MEA01]:

S1: mapping of evaluation of event types to tasks

S2: scheduling and invocation of event composition as well as delivery of results

S3: a mechanism in which the event composition algorithm is implemented

S4: a subscription service that manages connections between event types and contributing event types (implying both inter-task as well as intra-task communication) as well as subscribers

S5: memory management

S6: timeout service

In addition, this section addresses the following two issues:

1. event parameters that affect memory management as well as logical event occurrences Def. 4.12 on p. 50

2. event criticality that affects mapping of event types to tasks as well as subscription
4.9 Architectural Issues of Event Detection

4.9.1 Mapping Event Types to Tasks

It is possible to partition event types to different tasks performing the event composition. There are two extreme cases. One extreme is to let one task perform composition for all event types. The other extreme is to let each event type be managed by one or more tasks each.

Event specification languages can include specification of tasks. The advantage of avoiding a task specification is that it is possible to post-partition the event types onto tasks, whereas the usage of task specifications limits this ability. The advantage of task specifications is that it is possible to apply mechanical reasoning to the specification with respect to tasks. For example, it is possible to check if event monitoring is schedulable together with the rest of the task load.

None of the addressed event specification languages has explicit task specification except Solicitor (in Ch. 10). TCCS has an implicit task specification that partitions each event type in one or more tasks each.

4.9.2 Invocation, Scheduling, and Delivery

Two related issues are invocation and scheduling of event composition. Event composition may be invoked implicitly, after each primitive event occurrence has been signalled, or explicitly, for example, at fixed periods of time or when all primitive event occurrences occurring in the same clock granule have been delivered.

Scheduling of event composition may be data-driven or control-driven. Data-driven scheduling is proposed in Snoop (in [CM94, CKAK94]). The data-driven scheduling fits the (recursive) procedure call semantics and, thus, requires less code to implement compared to control-driven scheduling. In control-driven scheduling, a scheduler invokes evaluation of each event type. Depending on the result of the evaluation of an event type, evaluation of dependent event types may have to be scheduled for evaluation. Jaeger [Jae97] proposed parallel event composition based on data-driven scheduling.

Finally, delivery to subscribers may be immediate as proposed in Snoop [CM94, CKAK94] or delayed until all results stemming from event occurrences in an evaluation are processed (analyzed). Similarly to data-driven scheduling, immediate delivery fits the semantics of procedure calls and, hence, requires less code than delayed delivery. However, immediate delivery does not allow the subscriber to schedule their work depending on the result of event composition, since, in the view of the subscriber, a more im-
portant event occurrence may arbitrarily be delivered after a less important event occurrence. Therefore, delayed delivery is preferable in, for example, real-time systems where it is important to avoid arbitration, such as the subscriber taking an incorrect action due to that the resulting event occurrences do not arrive at the same time.

Coupling modes (in Section 3.4.1 on p. 34) are not, in themselves, related to the kind of invocation that is employed, since coupling modes only relate event composition, condition evaluation, and action execution of an ECA-rule to the scope of transactions; the only requirement is that event composition is invoked within the scope of a transaction.

Likewise, coupling modes are not, in themselves, related to the delivery of the results; the only requirement on event composition is that the results are delivered to the same transaction that generated the event occurrences in immediate and deferred coupling modes.

4.9.3 Event Composition Mechanism and Event Subscription

The event composition mechanism and subscription mechanism are closely related, since intra-task subscription is dependent on the event composition mechanism. For example, places in Petri-nets are used to combine Petri-nets representing the event operators. In Table 4.8 on the facing page, the event composition mechanisms of the event operators in the different event specification languages are presented. In Table 4.9 on p. 80, the available intra-task subscription mechanisms are presented.

Inter-task subscription depends on other mechanisms such as shared variables (e.g., Hoare’s monitors) or message passing (e.g., asynchronous message passing, remote procedure calls etc.). In most cases, an event bus (in Section 2.1.2 on p. 15) based on message passing is used.

In Snoop, Fig. 4.6 on the facing page depicts the structure of the event operators and how they are connected to each other for the event expression \((E_1; E_2)\Delta E_3\). Each operator and primitive event type is an object where the buried direct pointers means that the node representing \(E_1\) and \(E_2\) points to the principal operator that is a sequence. Event occurrences are signalled with either asynchronous message passing or (remote) procedure calls depending on if the subscription is intra-task or inter-task. The definition of the operator is procedural. For example, Adaikkalavan [Ada02] presents this as in Fig. 4.7 on p. 81. This procedure is called for each contributing event occurrence when it is signalled.
4.9 Architectural Issues of Event Detection

<table>
<thead>
<tr>
<th>Language</th>
<th>Operator mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solicitor</td>
<td>Procedural</td>
</tr>
<tr>
<td>Snoop</td>
<td>Procedural</td>
</tr>
<tr>
<td>SAMOS</td>
<td>Petri-nets</td>
</tr>
<tr>
<td>ODE</td>
<td>Finite State Automata</td>
</tr>
<tr>
<td>TCCS</td>
<td>?</td>
</tr>
<tr>
<td>RTL</td>
<td>Simple temporal networks</td>
</tr>
<tr>
<td>Chronicle recognition</td>
<td>Simple temporal networks</td>
</tr>
<tr>
<td>Interval-based Event Algebra for Restricted Detection</td>
<td>Procedural</td>
</tr>
<tr>
<td>EPL</td>
<td>Condition-action rules&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<sup>a</sup>Prolog-like condition action rules.

Table 4.8: Operator mechanism for event specification languages

In SAMOS, Fig. 4.8 on p. 82 depicts the \((E_1;E_2)\triangle E_3\) expression as a Petri-net instead. If the operators are in processed in a single task, then the places between the event operators are ordinary places in a Petri-net. Otherwise, if the event operators are processed in different tasks, then these places would be represented by some mechanism based on asynchronous message passing semantics.
In ODE, Fig. 4.9 on p. 82 depicts $(E_1;E_2) \Delta E_3$ using finite state automata. The example assumes that the sequence and the conjunction operator are managed by two different tasks. By following StateCharts [HPSS87] semantics, an event occurrence is signalled to every state machine in the system. N.B., the syntax $E_1/E_2$ on an outgoing edge from a node means that if $E_1$ has occurred then an $E_2$ occurrence is generated.

Monitoring of real-time logic and Chronicle recognition both uses directed weighted graphs to represent an event type, where the vertices represent event types and the edges represent time constraints as well as relative order of occurrence. For example, Fig. 4.10 on p. 83 is a depiction of $(E[10 \leq |\text{span}(E)| \leq 20])!$ where $E = ((E_1!):(E_2!))!:E_3!$ in Chronicle recognition. By default, the constraint is $[-\infty, \infty]$. This constraint graph is processed by a Floyd-Warshall-like algorithm to obtain all shortest pairs. During event composition, each node is associated with a time

<table>
<thead>
<tr>
<th>Language</th>
<th>Intra-task subscription</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snoop</td>
<td>Buried direct pointers, connector sets?</td>
</tr>
<tr>
<td>Snoop</td>
<td>Buried direct pointers</td>
</tr>
<tr>
<td>SAMOS</td>
<td>Places</td>
</tr>
<tr>
<td>ODE</td>
<td>State change or signal</td>
</tr>
<tr>
<td>TCCS</td>
<td>?</td>
</tr>
<tr>
<td>RTL</td>
<td>Edges in simple temporal network</td>
</tr>
<tr>
<td>Chronicle recognition</td>
<td>Edges in simple temporal network</td>
</tr>
<tr>
<td>Interval-based Event Algebra for Restricted Event Detection</td>
<td>Variables and queues</td>
</tr>
<tr>
<td>EPL</td>
<td>Tables in a deductive database</td>
</tr>
</tbody>
</table>

Table 4.9: Existing composition mechanisms in the event specification languages

*aIn data-driven scheduling of event composition buried direct pointers are used, whereas in control-driven scheduling of event composition connector sets are used.*
procedure seq_general(event_instance, parameter_list) begin
  if event_instance is the left event begin
    append event_instance to leftQueue(E)
  end
  if event_instance is the right event begin
    foreach e in leftQueue(E)
      if timestamp(e) > timestamp(event_instance) then
        signal ⟨event_instance, e⟩ to parent
      end
  end
end

Figure 4.7: Procedural definition example in Snoop [Ada02]

window initially set to \([-∞, ∞]\) and each line of reasoning is stored in a set of \(L = \{⟨E, [t, t']⟩\}\) structure. All lines of reasoning are stored in a set \(C = \{L\}\). An event occurrence can be integrated into a line of reasoning if the termination time of the event occurrence is within the time window of the node representing the event type. This is done by copying the line of reasoning and propagating the time of occurrence of the event occurrence into the copy. In other words, for each event occurrence and each line of reasoning where this event occurrence can be integrated, there is a new line of reasoning added to \(C\). This propagation algorithm [Dou96] has a time complexity of \(O(m^2)\) where \(m\) is the number of event types in the Chronicle. They claim that on average the time complexity is \(O(Km^2)\) where \(K = |C|\). However, since \(|C|\) is dependent on the processing, this is a simplification that does not define the upper bound time complexity.

In the approach of monitoring of RTL expressions (e.g., Chodrow et al. [CJD91]), the algorithm is similar to Chronicle recognition with the following constraint: the order of the event occurrence according to a filtered event log is used to check if it can be integrated into a line of reasoning. As a result of this constraint, it is possible to reduce the complexity of the propagation of time of occurrence to \(O(n)\) [ML97a, ML97b], where \(n\) is the number of basic RTL expressions in a constraint, when the \# function is not used in
82  Event Composition: Syntax, Semantics, and Processing

Figure 4.8: Petri-net representation of \((E_1; E_2) \triangle E_3\) [Gat94, pp. 101-102]

Figure 4.9: Finite state automata representation of \((E_1; E_2) \triangle E_3\)

an RTL expression. If the function \(\#\) is used, then the complexity is \(O(n^3)\) according to Liu et al. [LMK98].

4.9.4 Memory Management

The memory management for event composition encompasses two parts: (i) the memory required for event expressions (e.g., objects representing event operators and subscription), and (ii) the memory required to keep data
representing event occurrences. Part (i) is only used for management of event monitoring. Part (ii) is used when the event monitor is operational.

In no supporting system for an event specification language is memory management (for part (ii)) addressed except by Mellin [Mel00]. By considering the various requirements for on-line monitoring, the overhead for memory management can be reduced significantly as reported in Part III. Event parameters affect the memory management of event occurrences.

### 4.9.5 Event Parameters

Each primitive event type carries a set of fixed system parameters that are used to determine the specifics about an event instance. A minimal set in synchronous monitoring (see Section 2.1.4 on p. 16) is the event type and termination time. If asynchronous monitoring is employed, the temporal and logical scope (event scope) in which the event was generated is desirable to include. For example, in SAMOS the user and transaction identifier are included among the parameters to enable parameterized event operators. At least the transaction identifier is required in the asynchronous monitoring case for active databases. The system parameters required for synchronous monitoring is subsumed by those required for asynchronous monitoring.

Composite event occurrences derive system parameters from their contributing event occurrences. This derivation can either be explicit or implicit. Explicit derivation requires a specification addressing how parameters should be derived when composition takes place. Moreover, it may be desirable to perform simple computation. In Chronicle recognition [Dou96] as well as Java event monitor (JEM) [LM99] for monitoring of real-time logic expressions, this explicit derivation is part of the action part. In Snoop, SAMOS, ODE, HiPAC, TCCS this derivation is implicit; for example, the timestamp of the composite event occurrence is assigned to the timestamp of the terminator event occurrence.
Event Parameter Conditions: The semantics of event parameter conditions is captured in the logical event occurrence (in Def. 4.12 on p. 50) where $Q$ only addresses event parameters. It can be useful to perform condition evaluation during event composition instead of leaving all condition evaluation to a later stage. For example, assume that a temperature sensor sends temperature reading events periodically, but only if the temperature is above $95^\circ$ Celsius is an event occurrence critical. Instead of sending each event occurrence for evaluation by another software component, the non-critical occurrences can be filtered out. The reason for the added usefulness of separate event parameter conditions is that: (i) the additional semantics allows us to model composite events across transaction scopes better and, thereby, handle otherwise unmanageable situations (e.g., only compose events out of contributing event occurrences raised (indirectly) by the same user irrespective of transaction scope); and (ii) more events can be filtered out at an early stage so that less load is imposed on the system. In addition, the same mechanism can be used for more efficient support of relative and absolute temporal validity notions that are important for real-time applications [Ram93].

Event parameter conditions were introduced in ODE as a way of removing the condition altogether. However, this implies a rich semantics in unrestricted event parameter conditions, which is a source of unpredictability (and inefficiency). Therefore, this approach is not assumed in this Ph.D. thesis. Given that the data types of the parameters are restricted to scalar types, as in SAMOS\(^7\), the condition evaluation can be sufficiently efficient for many applications. Berndtsson [Ber98] applied event parameter conditions to event graphs in active databases and argues that a part of the condition of an ECA rule can be processed during event composition rather than the condition evaluation.

### 4.9.6 Timeout Service

A timeout service is required to enable the $E+T$ event. One important issue is to reduce the number messages containing virtual timeouts generated by a physical timeout. In theory, there may be a timeout for each monitored $E_i+T$ at each (point in) time. For example, assuming there are $n$ $E_i+T$ specification, then there can be $n$ virtual timeouts for each (point in) time.

\(^7\)Only system parameters are allowed.
All relative temporal event expressions monitored by the same task only need one timeout message instead of one message for each virtual timeout in the monitored relative temporal event expressions. This message should preferably contain identities of which relative temporal event expressions that have occurred.

4.10 Event Criticality

Berndtsson and Hansson [BH95] have addressed the event criticality issue. This affects both scheduling of task performing event monitoring, subscription as well as timeout services. In essence, critical event occurrences must be prioritized and treated before less critical event occurrence. This has two implications. First of all, event types of different criticality should not be placed into the same task, since it can have an adverse effect on achieving timeliness. For example, if a task is composing an event occurrence of low criticality and a high critical event occurrence is signalled to this task, then the composition of the high critical event composition is delayed by the low critical event composition. Secondly, some kind of management of different urgency levels must be supported by the inter-task subscription mechanism; for example, the underlying network must support different levels of latency classes to manage event occurrences of different criticality. Otherwise, critical event occurrences may be delayed for an unpredictable duration.

Given that critical event occurrences are prioritized, the filtered event log may become unordered. As Berndtsson and Hansson [BH95] remarked, this is not acceptable and they proposed to add criticality to event expressions (either implicitly derived from time constraints of triggered tasks or explicitly specified). In effect, a composite event type must be based on event types of the same criticality. Therefore, each event occurrence must be duplicated for each criticality it has to maintain order of occurrence. Berndtsson and Hansson proposed to use logical events (in Def. 4.12 on p. 50) to specify event types of different criticality. For example, if the event tempReading is used in tempReading;pressureReading of high criticality and as tempReading/logTimeEvent of low criticality, then we would use tempReading[true](criticality := high) instead of tempReading in the first expression and tempReading[true](criticality := low) in the second expression. If tempReading only is critical if the temperature is above 95°C Celsius, then the critical expression can be written critTempReading := tempReading[temperature ≥ 95](criticality := high).
Chapter 5

Problem Definition

“The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.”

/Theodore Rubin

This chapter presents the purpose, the motivation, the assumptions and their implications, the strategy and goals, and the key hypotheses of this work.

5.1 Purpose

The purpose of this work is, first and foremost, to impart general principles for achieving resource-predictable monitoring of events. Moreover, its purpose is also to show general principles for achieving efficient event monitoring. The reason for these purposes is that achieving timeliness (in Section 3.1.1 on p. 27) requires all tasks (including tasks performing event monitoring) in a system to have known bounded resource requirements (such as processor time, memory usage, etc.) as well as being sufficiently efficient. Sufficient efficiency means that the entire task load must be schedulable to meet given time constraints.
5.2 Motivation

There are several systems that use event monitoring with stringent response time requirements. For example, on-line monitoring for on-line usage can be found in active (real-time) databases (in Section 3.4 on p. 33), fault management (in Section 3.3 on p. 32), and the control loop of real-time systems (in Section 3.1 on p. 26). Event composition is a useful processing technique in event detection for on-line usage, since event composition can analyze a filtered event log in a more efficient manner than condition evaluation of a database state (e.g., as in condition evaluation of condition-action-rules). The main reason for the improved efficiency is that in event composition only a fraction of the database state (i.e., a partial view of the current or past state) may have to be analyzed compared to analyzing a significant part of the database state.

5.3 Assumptions and their Implications

Fig. 5.1 on the facing page (based on Fig. 2.1 on p. 13) depicts the assumed event monitoring functionality (where the addressed activity composition is emphasized). All functionality (data collection, event composition, action execution) are assumed to execute in tasks on the same resources as the (tasks of the) monitored object. Concerning data collection, the push protocol is emphasized, however, all results in this work are applicable to a pull protocol. N.B. this assumption also covers configurations where event monitoring is allocated to a dedicated processor (e.g., Mellin et al. [MHA97], Mellin [Mel98a], Hansson [Han99]).

Since efficiency cannot be studied in general, but only with respect to specific requirements and given resources, algorithmic time complexity is investigated, since it defines how well the response time of algorithms scale with respect to the size of the input (e.g., the size of the filtered event log). In other words, it is possible to tell how the response time behaves with respect to the size of the input. An algorithm that scales worse than another algorithm require more resources to perform the processing given a sufficiently large input size. Algorithmic time complexity can, at least, guide us in the choice of contending algorithms and configurations (of algorithms) when the size of the input grows.

It is useful to investigate the time complexity of event monitoring in real-time systems, even though time complexity analysis may be of little or no use
5.3 Assumptions and their Implications

To real-time systems in general, since many real-time systems are designed in such a way that we do not reach sufficiently large sizes of inputs to make the results meaningful. This investigation is useful for small input sizes, since we analyze different configurations of the same (abstract) algorithm (presented in our formalized schema) rather than comparing different algorithms solving the same problem. Moreover, one of the application areas is active real-time databases, in which the filtered event log may grow to a size where the time complexity is useful in general.

![Figure 5.1: Assumed event monitoring structure](image)

Event composition and management of the filtered event log are the sources of unpredictability in event monitoring, since their resource requirements can grow indefinitely, unless event composition processing is constrained.

Event monitoring in this work is based on asynchronous monitoring (in Section 2.1.4 on p. 16) outside the scope of transactions (in Section 3.4.1 on p. 34). The number of subscribers of an event type is limited by the (system) specification to enable event signalling with predictable resource requirements (in Section 2.1.2 on p. 15). Concurrent event occurrences are avoided (in Section 4.6 on p. 63), since it is a computationally cost-effective method to manage them for real-time systems. N.B. this avoidance does not affect the results of this work, since other management of concurrent events leads to the same results (in terms of resource-predictability as well as time complexity) but a computationally more costly way of performing event composition.
5.4 Strategy and Goals

The main strategy is to address the problem of timeliness in three themes: theme 1 named “Formal issues: Resource-predictable event monitoring” addressing resource-predictable monitoring of events as well as algorithmic time complexity, theme 2 named “Architectural issues: Efficient event monitoring” that address efficient designs for monitoring of events, and theme 3 named “Results” that address validation of the accuracy of the theoretical models constructed in theme 1 as well as a discussion on resource-predictable and efficient event monitoring.

In theme 1, the strategy for achieving resource-predictable event composition is to introduce a formal definition of a declarative event specification language (named Solicitor [Mel98a]), in which event specifications can be compiled into resource-predictable and efficient event monitors. Since management of temporal constraints are the method used in real-time systems to achieve timeliness, the main method is to add a minimum set of these temporal constraints to event expressions to achieve resource-predictable event composition.

To address the existence of a bound on the number of steps an event composition algorithm has to perform and (time) complexity of such an algorithm, it is necessary to formally define event operators, event contexts, and event composition.

Theme 2 addresses architectural issues of event composition, since there is insufficient material covering event monitoring for (embedded) real-time systems. Such an architecture is also necessary when discussing efficiency of implementations.

As mentioned, theme 3 encompasses formal1 experiments whose purpose is to validate the accuracy of the theoretical models constructed in theme 1 as well as discussion and related work on resource-predictable and efficient monitoring of events.

The goals of this work are as follows:

G1: Theme 1: Formal issues: Resource-predictable event monitoring

G11: To provide a stringent formal schema for the specification of declarative event specification languages where event expressions can result in resource-predictable event monitors. This

---

1A formal experiment is formal in the sense that it is structured and systematic.
5.4 Strategy and Goals

step is necessary to ensure resource-predictability. This formal schema has the following requirements:

- The specification of event operators should be parameterized on the specification of event context in order to enable reasoning about all event operators for an event context without having to consider all possible event operators and vice versa.
- The event contexts should be formally defined.

G12: To demonstrate the strength of the formal schema by
  - Specifying an event specification language (Solicitor) within the schema
  - Demonstrating that this event specification language can be used for realization of resource-predictable event composition algorithms

G13: To prove that an event monitor generated by an event expression in the event specification language has a bound on its resource requirements in terms of processor time (and memory requirements) regardless of event context.

G14: To investigate the algorithmic time complexity of processing arbitrary event expressions with arbitrary event histories in the event specification language.

G2: Theme 2: Architectural issues: Efficient event monitoring

G21: To specify a software architecture of an event monitor that meets the requirements of (distributed) real-time systems.

G22: To suggest appropriate design choices for real-time systems.

G23: To design an improved memory management for event monitoring that meets the specific requirements of (distributed) real-time systems. The main requirement is that in asynchronous event monitoring, the resulting composite event may need to be copied and multiplied to different address ranges.

G3: Theme 3: Results

G31: To validate the model of event composition derived from the formal schema (defined for meeting goal G14) by experimenting with algorithmic time complexity
Problem Definition

**G32:** To compare and contrast this work with other work

Distribution aspects are not considered. However, the proof of the bounded resource requirements of event composition algorithms is valid for distributed systems and the work on algorithmic time complexity gives, at least, an upper bound on the worst-case algorithmic time complexity for distributed systems.

Although the distribution aspects are disregarded, the intent is to enable the formalism and the architecture to handle necessary extensions for distribution. The approach is to avoid optimizations for the centralized case (e.g., assuming that shared memory is available for an implementation of the filtered event log management).

The key hypotheses of this work are as follows:

**Resource-predictability hypothesis:** It is possible to construct event composition algorithms that enable monitoring of any event expression in any event context with predictable resource requirements.

**Scalability hypothesis:** Given the resource-predictability hypothesis, it is possible design the aforementioned event composition algorithms to scale polynomially in time with respect to all significant input domains.
Part II

Semantics of Event Composition
This part is theme 1 and addresses formal issues of the problem by defining the semantics and processing of event specification languages in a precise and unambiguous way. Ch. 6 contains a formalized schema for event specification languages covering event semantics including processing fulfilling goal G11. In other words, event contexts can be formally defined in an event specification language. In contrast to other schemas (such as monitoring of real-time logic), this schema is parameterized in such a way that it is possible to reason about processing for an event operator in any event context or for all event operators in an event context. This simplifies the inclusion of new event operators as well as new event contexts. The event specification language Solicitor defined by Mellin [Mel98a] (presented in Ch. 4) has been refined and formally specified using this schema. The semantics of Solicitor is fully defined in Appendix B. In Ch. 7, it is demonstrated that it is possible to translate the formalism into a procedural algorithm (fulfilling goal G12). Ch. 8 contains a proof for all event expressions in all event contexts that there are predictable resource requirements on event composition given appropriate temporal constraints (fulfilling goal G13). Given this proof of existence of upper bound resource requirements, Ch. 9 addresses the time complexity issue (fulfilling goal G14).
Chapter 6

Formal Specification of Event Composition

“Eureka!”
/Archimedes

“Lasciate Ogne Speranza, Voi Ch’Intrate”
/Dante Alighieri

6.1 The Basis for an Event Composition Schema

An introduction to the formal notation for event specification languages and their semantics (and pragmatics) was presented in Section 4.5.1 on p. 52 for the purpose of describing and comparing. This formal notation is refined in this chapter, for the purpose of meeting goals G11 on p. 91 (in Section 6.1 to Section 6.2 on p. 102) and G12 on p. 91 (in Section 6.3 on p. 120). The complete formal notation and symbol dictionary is presented in Appendix A.

Axioms 6.1 to 6.2 on pp. 97–98 are the foundation for our formalism.

Axiom 6.1
Primitive event occurrences are instantaneous:

∀ γ ∈ G (prim(type(γ)) ⇒ start(span(γ)) = end(span(γ)))

□
In other words, all primitive event occurrences (in the generated set $G$) are instantaneous and thus atomic. As mentioned earlier, this is similar to the definition by Chakravarthy et al. [CM94] who state that all event occurrences are instantaneous and atomic (in Snoop with termination semantics).

**Axiom 6.2**

**Primitive event occurrences of the same type are not simultaneous:**

$$\forall \gamma_1, \gamma_2 \in G \ (\text{prim} (\text{type}(\gamma_1)) \land \text{type}(\gamma_1) = \text{type}(\gamma_2) \land \gamma_1 \neq \gamma_2 \Rightarrow \text{span}(\gamma_1) \neq \text{span}(\gamma_2))$$

□

The reason for Axiom 6.2 is that simultaneous event occurrences of the same event type would complicate event composition specification since indeterminism is introduced. The rationale is that primitive event occurrences of the same event type can be assigned discriminating timestamps.

### 6.1.1 The Problem of Event Contexts

The semantics of event operators in Section 4.4 on p. 46 does not include event contexts [Ada02]. These contexts, when different from general event context, are captured in the algorithm of the operator. Each combination of operator and context has its own algorithm. For example, given that each operator is realized as a process, then the algorithm for the sequence operator in chronicle context can be specified in pseudo code as in Fig. 6.1 on the facing page.

Let $O$ be a set of event operators and $C$ be a set of event contexts, then there is an algorithm for each combination ($O \times C$). To prove a property such as resource-predictability of an event context, it has to be proven for significant event expressions combining event operators (in $O$). If another event operator is added (to $O$), then the number of possible combinations of event expressions increase. To prove a property requires case-based proofs as well as proof by induction. It would be better if addition of a new event operator or a new event context would not increase with the number of operators or contexts. Further, the less cases there are to manage in a proof, the more tractable the proof is.
Let $E = E_1; E_2$ in

process sequence<$E$>
var
  $Q_1$: set<$E_1$>
  $e, etmp$: any_of<$E_1, E_2$>
  $cev$: $E$
while true do
  /* receive signalled event occurrence, subscription implicitly derived from event specification of $E$ */
  receive $e$ where type($e$) $\in \{E_1, E_2\}$ /* receive signalled occurrences*/
  if type($e$) = $E_1$ then
    $Q_1 := Q_1 \cup \{e\}$ /* add $E_1$ occurrence */
  else /* type($e$) = $E_2$ */
    if not empty($Q_1$) then /* there is an $E_1$ occurrence */
      /* select earliest available $E_1$ occurrence (view set as a queue): */
      $etmp := \text{earliest}(\text{sortAscendingOnTerminationTime}(Q_1))$;
      $cev := \text{combine}(etmp, e)$; /* compute composite event occurrence */
      $Q_1 := Q_1 \setminus \{etmp\}$; /* consume selected $E_1$ occurrence */
      postEventForSignalling($cev$);
    endif
  endif
endwhile
done

Figure 6.1: Sequence operator algorithm for chronicle event context

For example, given two event contexts (recent and chronicle) and two event operators (sequence and conjunction), there are 4 basic cases of expressions containing two contributing event types. To ensure the proof by induction, the composition of contributing event types must be demonstrated to be resource-predictable as well as addressing how cyclical specifications (if such are allowed) are managed. Adding another event operator increases the number of basic cases to 6 (two more proofs must be included as well as checked that the proof by induction holds). Adding another event context increases the number of basic cases to 9. A better way is to be able to prove a property for an event context for all event operators given that they follow a set of well-defined constraints.

Another example is testing a statement: in general context the expression $(E_1!) \triangle (E_2!)$ results in the same set of occurrences as the event expression
((E_1!); (E_2!)) \lor ((E_2!); (E_1!)). However, this is not true in, for example, the chronicle context. This expression must be studied for each context separately to demonstrate that the results are equal.

It would be better to study what properties of event contexts are associated to specific algebraic properties such as reflexivity, symmetry, transitivity, and distributivity. In the example in this paragraph, reusability of event occurrences for different composite event occurrences is a necessary requirement for the transformation to be legal.

There have been a few attempts to formalize event composition (e.g., [Jae97, CYY98, LMK98, Ada02]). For example, the formalization of Snoop by Chakravarthy et al. [CYY98] is defined in terms of the general context. Then, they define a filter for each context that defines which composite event occurrences that have occurred according to the context. This is a compelling idea, however, they need to make some event occurrences carry more information than necessary to make the formalism work. For example, event occurrences generated by the aperiodic event expression carry event occurrences of both E_1 and E_2 types (from the aperiodic expression in Def. 4.8 on p. 48) instead of only E_2 event occurrences that would match the operator semantics. This formalization by Chakravarthy et al. [CYY98] was not chosen for this work, since it does not contain information about the processing itself. Only the outcome of the processing is emphasized. A more detailed comparison is found in Section 6.5.1 on p. 143.

Another attempt of formalizing Snoop is by Adaikkalavan [Ada02], who bases his work on Galton et al. [GA01]. However, processing is not considered in this formalization either.

A basic problem in formalization of event composition is that first-order predicate logic cannot handle consumption of truths. This has been addressed in the field of linear logic (e.g., Wadler [Wad93]). This can be used to state that “if I have $10 I can either buy a pizza or a hot dog but not both”. In traditional (intuitionistic) logic, if there is a fact that “I have $10”, then this truth can be used in an arbitrary number of resolutions. In a sense, it would be possible to buy any number of pizzas and hot dogs with the same $10.

However, in contrast to linear logic that consumes truths in general, event composition processing uses or consumes event occurrences with respect only to an event type. Reuse of event occurrences is allowed in some event contexts according to the rules of the event context. That is, the same principal event operator of an event type in different event contexts can use contributing event occurrences while allowing or disallowing reuse.
6.1 The Basis for an Event Composition Schema

To resolve the problems of use, reuse, and consumption, we have chosen to introduce restricted production rules for the event operators defined in first-order predicate logic (in Def:s 4.3 to 4.11 on pp. 47–49). There are two advantages: (i) processing of production rules is an iteration [Nil82, p. 21] that can be analyzed in terms of, for example, time complexity; and (ii) it is possible to restrict these rules in such a way that we can meaningfully transform rules representing event expressions into other rules by symbolic manipulation to, for example, check the equivalence of two event expressions. N.B., although the formal schema is based on condition-action rules, the intention is not to implement event composition in terms of such rules.

Production rule systems (also known as production systems) have been addressed in artificial intelligence (e.g., seminal work by Nilsson [Nil82]). A production system architecture consists of three main components [Nil82, p. 17]: (i) a rule base (set of production rules); (ii) an inference engine (realizing a control strategy); and (iii) a (global) database. The outline of an algorithm processing rules is depicted in Fig. 6.2

\[
\text{DATA} \leftarrow \text{initial state} \\
\text{until } \text{DATA} \text{ satisfies the termination condition, } \text{do begin} \\
\quad \text{select some rule, } R, \text{ in the set of rules that can be applied to } \text{DATA} \\
\quad \text{DATA} \leftarrow \text{result of applying } R \text{ to } \text{DATA} \\
\text{end}
\]

Figure 6.2: Basic production system algorithm [Nil82, p. 21]

The \text{DATA} in our schema consists of event types as well as event occurrences. The rule base (i) contains production rules defining the semantics of event operators. Our restriction of these production rules is that the conditions are written in first order predicate logic and the rules can only assign sets of event occurrences that are checked by the conditions; there are general sets (such as \(G\)) as well as specific sets associated with event types (e.g., a set can contain the event occurrences that have been used to form a composite event occurrence of an event type). The processing is similar to forward deduction [Nil82, section 6.1], where event occurrences can be viewed as facts and composite event occurrences can be viewed as deducing new facts. Similarly to forward deduction, when no new composite event occurrences can be found (no new facts can be deduced), processing terminates.
The control strategy, managing conflicts in the selection of a production rule to that will be evaluated (component (ii)), may either be between production rules representing an event type or between production rules representing different event types. The former conflicts are managed by writing the conditions in such a way that conflicts are avoided. The latter conflicts are managed by scheduling the evaluation of the production rules so that contributing event types are processed completely before a contributed event type. This scheduling is possible, because recursive event types such as $E = E; E_1$ are disallowed. Further, in terms of production systems, composite event occurrences are irrevocable facts; that is, once a composite event occurrence is formed it is never removed.

With respect to the database (component (iii)), event composition is assumed to take place in a partitioned memory. For example, results may be replicated to different physical nodes in a distributed system.

6.2 The Event Composition Schema

Let the sequence of states $s_0, s_1, \ldots, s_n$ represent the history of states of event composition, where each $s_i$ is based on the generated set $G$. Each update of $G$ constitutes a transformation from $s_i$ to $s_j$ where $i \neq j$. One such type of transformation is the addition of (tokens representing) primitive event occurrences to $G$. Event composition takes place after these primitive event occurrences have been added; either immediately after each primitive event occurrence or delayed to some meaningful time where the primitive event occurrences are stable (i.e., no arriving event occurrences can precede the event occurrences that have arrived, cf. Section 4.6 on p. 63). Event composition is a sequence of state transformations that ends when all new event occurrences have been processed. This processing is followed by signalling composite event occurrences to subscribers of each event type.

The basis for these state transformations are the event types. In our schema, each event type maintains a history of event occurrences they have used and invalidated represented as sets. The result of an operator is based on the generated set $G$ and these histories of used or invalidated event occurrences. To compare and contrast different event contexts with respect to invalidation etc., consider Example 6.1 on the facing page depicted in Fig. 6.3 on the facing page that address recent and chronicle event contexts.
6.2 The Event Composition Schema

Example 6.1
Assume \( G = \{ \Gamma(E_1, [1,1]), \Gamma(E_2, [2,2]), \Gamma(E_2, [3,3]), \Gamma(E_2, [4,4]), \Gamma(E_2, [5,5]) \} \).
If we monitor \( E_{12} = E_1; E_2 \), then \( \text{occ} \) predicate holds true for \( \{ \Gamma(E_{12}, [1,3]), \Gamma(E_{12}, [2,3]), \Gamma(E_{12}, [1,4]), \Gamma(E_{12}, [2,4]), \Gamma(E_{12}, [1,5]), \Gamma(E_{12}, [2,5]) \} \) (in general context). Further, let \( E_{12r} = E_1; (\text{context: recent})E_2 \) be the corresponding event specification in recent context and \( E_{12c} = E_1; (\text{context: chronicle})E_2 \) in chronicle context.

![Diagram showing event occurrences and their contexts](image)

Figure 6.3: Depiction of Example 6.1 in general, recent, and chronicle event context

In Example 6.1, the following event occurrences are used by the sequence operator according to the contexts: in recent context \( \Gamma(E_1, [2,2]) \) and \( \Gamma(E_2, [3,3]) \) forms the event occurrence \( \Gamma(E_{12r}, [2,3]) \), \( \Gamma(E_1, [2,2]) \) and \( \Gamma(E_2, [4,4]) \) forms \( \Gamma(E_{12r}, [2,4]) \), and \( \Gamma(E_{12r}, [2,5]) \) is formed by \( \Gamma(E_1, [2,2]) \) and \( \Gamma(E_2, [5,5]) \); in chronicle context the occurrences \( \Gamma(E_1, [1,1]) \) and \( \Gamma(E_2, [3,3]) \) form \( \Gamma(E_{12c}, [1,3]) \), and, finally, \( \Gamma(E_{12c}, [2,4]) \) is formed by \( \Gamma(E_1, [2,2]) \) and \( \Gamma(E_2, [4,4]) \). N.B., the recent context allows reuse, whereas chronicle context does not. Additionally, in recent context \( \Gamma(E_1, [1,1]) \) is not selected (im-
licitly invalidated) since $\Gamma(E_1, [2,2])$ is more recent and in chronicle context $\Gamma(E_2, [5,5])$ is invalidated by the sequence operator since it cannot contribute to a future occurrence of $E_{12c}$.

As is discussed in Section 4.5.2 on p. 54, it is useful to distinguish between initiator and terminator occurrences of composite event types. Therefore, the subscript ‘$\alpha$’ denotes initiators and ‘$\omega$’ denotes terminators.

Ordered pairs (binary tuples) $\langle \gamma', \gamma \rangle$ are used to define the relation between initiators or terminators $\gamma'$ and the composite event occurrence they have initiated or terminated $\gamma$.

**Definition 6.1**
**Initiator or terminator-set-of function:**
Let $\mathcal{U} = \{\langle \gamma', \gamma \rangle\}$ (where $\mathcal{U}$ is any set of initiator/terminator tuples), then

$$\text{iot}(\mathcal{U}) = \{\gamma'|\langle \gamma', \gamma \rangle \in \mathcal{U}\}$$

□

**Definition 6.2**
**Composite-set-of function:**
Let $\mathcal{U} = \{\langle \gamma', \gamma \rangle\}$, then

$$\text{com}(\mathcal{U}) = \{\gamma|\langle \gamma', \gamma \rangle \in \mathcal{U}\}$$

□

For each event type, the following usage and invalidation set are used: $\mathcal{U}_\alpha(E)$ is a set of $\langle \gamma', \gamma \rangle$ for $E$, where $\gamma'$ is the used initiator; $\mathcal{U}_\omega(E)$ is a set of $\langle \gamma', \gamma \rangle$ for $E$, where $\gamma'$ is the used terminator; $\mathcal{I}_\alpha(E)$ is a set of invalidated initiator occurrences for $E$; and $\mathcal{I}_\omega(E)$ is a set of invalidated terminator occurrences for $E$. The sets $\mathcal{G}$, $\mathcal{U}_\alpha(E)$, $\mathcal{U}_\omega(E)$, $\mathcal{I}_\alpha(E)$, and $\mathcal{I}_\omega(E)$ are all monotonically growing over time. Further, $\forall E (\text{iot}(\mathcal{U}_\alpha(E)) \subset \mathcal{G} \land \text{com}(\mathcal{U}_\alpha(E)) \subset \mathcal{G} \land \text{iot}(\mathcal{U}_\omega(E)) \subset \mathcal{G} \land \text{com}(\mathcal{U}_\omega(E)) \subset \mathcal{G} \land \mathcal{I}_\alpha(E) \subseteq \mathcal{G} \land \mathcal{I}_\omega(E) \subseteq \mathcal{G})$. In other words, all used or invalidated event occurrences are a subset of all generated event occurrences $\mathcal{G}$. N.B. only invalidated event occurrences sets can be equal

---

1The letters ‘$\alpha$’ and ‘$\omega$’ are the first and the last letter in the Greek alphabet.
6.2 The Event Composition Schema

to $\mathcal{G}$, since a used event occurrence is associated with a composite event
occurrence also belonging to $\mathcal{G}$.

The processing tuple $P_i = \langle E, \mathcal{U}_\alpha(E), \mathcal{I}_\beta(E), \mathcal{I}_\omega(E) \rangle$ defines the
state of an event type $E$ of $s_i$. The processing tuple set $P_i = \{P_i\}$ for all
event types in step $i$. The state $s_i$ is the generated set in state $i$ ($\mathcal{G}_i$) and the
set of event type states. That is, $s_i$ is defined as the binary tuple: $(\mathcal{G}_i, P_i)$.

The two predicates in Def:s 6.3 to 6.4 on the current page are defined to
simplify other predicates, since they are used frequently.

**Definition 6.3**

Initiator predicate:

$$initiate(\gamma_1, \gamma_2) \overset{\triangle}{=} \langle \gamma_1, \gamma_2 \rangle \in \mathcal{U}_\alpha(\text{type}(\gamma_2))$$

□

In other words, *initiate* is true for any $\gamma_1$ that initiates $\gamma_2$. The type is
derived from the $\gamma_2$ occurrence.

**Definition 6.4**

Terminator predicate:

$$terminate(\gamma_1, \gamma_2) \overset{\triangle}{=} \langle \gamma_1, \gamma_2 \rangle \in \mathcal{U}_\omega(\text{type}(\gamma_2))$$

□

In other words, *terminate* is true for any $\gamma_1$ that terminates $\gamma_2$. Similarly
to *initiate*, the event type is derived from $\gamma_2$.

To explain the event composition algorithm assumed in this schema, Def:s 6.5 to 6.9 on pp. 106–108 are introduced. This event composition algorithm
is an abstraction applicable to all algorithms addressed in Ch. 4.
Definition 6.5
Event occurrence set relations:

\[ X \subseteq X' \triangleq X \subseteq X' \land \forall \gamma, \gamma' (\gamma \in X \land \gamma' \in (X' \setminus X) \Rightarrow \text{end}(\gamma) \leq \text{end}(\gamma')) \]
\[ X \cup X' \triangleq X = X' \]
\[ X \cup X' \triangleq X' \cup X \]
\[ X \cup X' \triangleq X' \cup X \]

In other words, the event occurrence set relations are restrictions of the ordinary set relations. These restrictions are based on the time span of the event occurrences. For example, \( X \subseteq X' \) means that all event occurrences in the difference \( X' \setminus X \) must succeed all event occurrences in \( X \) in terms of termination time. The termination time is used, because these definitions are used to define processing; the termination time of an event occurrence is the earliest time when the system is notified of its occurrence.

Definition 6.6
Processing tuple is of event type predicate:

\[ \text{event}_\text{type}_\text{of} (P, E) \triangleq \forall E', U, U', I, I' (P = \langle E', U, U', I, I' \rangle \Rightarrow E = E') \]

In other words, the predicate \( \text{event}_\text{type}_\text{of} \) is true if the processing tuple \( P \) belongs to event type \( E \).
Definition 6.7
Successive processing tuples for the same event type:

\[
\text{later\_processing\_tuple\_state}(P_i, P_j) \triangleq \\
\forall E, U_i, U'_i, I_i, I'_i, U_j, U'_j, I_j, I'_j \\
\langle E, U_i, U'_i, I_i, I'_i \rangle = P_i \land 
\langle E, U_j, U'_j, I_j, I'_j \rangle = P_j \Rightarrow \\
\begin{align*}
&\left( (U_i \supseteq U_j \land U'_i \supseteq U'_j \land I_i \supseteq I_j \land I'_i \supseteq I'_j) \lor \\
&\left( (U_i \supseteq U_j \land U'_i \supseteq U'_j \land I_i \supseteq I_j \land I'_i \supseteq I'_j) \lor \\
&\left( (U_i \supseteq U_j \land U'_i \supseteq U'_j \land I_i \supseteq I_j \land I'_i \supseteq I'_j) \lor \\
&\left( (U_i \supseteq U_j \land U'_i \supseteq U'_j \land I_i \supseteq I_j \land I'_i \supseteq I'_j) \right) \\
&\right) \\
&\right) \\
&\right)
\end{align*}
\]

\[\square\]

In other words, \text{later\_processing\_tuple\_state} is true if the processing tuples belong to the same event type and \( P_i \) is a successive state of \( P_j \). At least one of \( U_\alpha(E) \), \( U_\omega(E) \), \( I_\alpha(E) \), and \( I_\omega(E) \) must have changed between the two processing tuples to be of successive states.

Definition 6.8
Same processing tuple for an event type:

\[
\text{same\_processing\_tuple\_state}(P_i, P_j) \triangleq \\
\forall E, U_i, U'_i, I_i, I'_i, U_j, U'_j, I_j, I'_j \\
\langle E, U_i, U'_i, I_i, I'_i \rangle = P_i \land 
\langle E, U_j, U'_j, I_j, I'_j \rangle = P_j \Rightarrow \\
(U_i = U_j \land U'_i = U'_j \land I_i = I_j \land I'_i = I'_j)
\]

\[\square\]

In other words, if the sets of used and invalidated event occurrences are the same for two processing tuples, then these processing tuples represent the same state.
Definition 6.9
Later state than:

\[ \text{later\_state}(s, s') \triangleq \]
\[ \forall G_i, G_j, P_i, P_j \]
\[ (s = \langle G_i, P_i \rangle \land s' = \langle G_j, P_j \rangle \Rightarrow G_i \supseteq G_j \lor)
\[ (G_i = G_j \land \exists E \forall E' (E \neq E' \Rightarrow (\forall P_i, P_i', P_j, P_j'
\[ (P_i, P_i' \in P_i \land \text{event\_type\_of}(P_i, E) \land
\[ \text{event\_type\_of}(P_i', E') \land
\[ P_j, P_j' \in P_j \land \text{event\_type\_of}(P_j, E) \land
\[ \text{event\_type\_of}(P_j', E') \Rightarrow
\[ \text{later\_processing\_tuple\_state}(P_i, P_j) \land
\[ (\text{later\_processing\_tuple\_state}(P_i', P_j') \lor
\[ \text{same\_processing\_tuple\_state}(P_i', P_j'))))))))} \]

In other words, if the \( G \) in state \( s \) is a superset of \( G \) in state \( s' \) then \( s \) is a later state than \( s' \), then the predicate is true; otherwise, if \( G \) in state \( s \) is equal to the \( G \) in state \( s' \), then the predicate is true if there is at least one event type whose processing tuple are successive in state \( s \) compared to state \( s' \) while all other event types’ processing tuples in \( s \) are equal or successive the processing tuples in \( s' \).

Each event operator is defined by a set of operator rules that transforms \( s_i \) into \( s_j \). When a set of events has occurred, the event composition algorithm defined in Def. 6.11 on the facing page is applied. This algorithm is the least constrained event composition algorithm in terms of conflict resolution among operator rules, since it arbitrarily selects an operator rule to evaluate. Before this definition, Def. 6.10 used in the definition is presented.

Definition 6.10
Rules function:
\[ \text{rules}(E) \text{ is a function that returns the set of rules } \{r\} \text{ applicable to } E. \]
Definition 6.11

Least constrained event composition: This algorithm is a refinement of the algorithm presented by Nilsson [Nil82, p. 21] depicted in Fig. 6.2 on p. 101. Let $s$ be the state of event composition; let the $\text{condition}(s, r, E)$ predicate be true if the condition of rule $r$ evaluated on event type $E$ is true in $s$ (precise definition in Def:s 6.15 to 6.16 on p. 113); let $\text{conclusion}(s, r, E)$ return a state change (precise definition in Def:s 6.15 to 6.16 on p. 113), let $I \doteq (\text{later}_\text{state}(s, s_0) \lor s=s_0)$, and let $C \doteq r \in \text{rules}(E) \land \text{condition}(s, r, E)$ then

```
while $\exists E, r (C)$ do begin
  $\{I \land \exists E, r (C)\}$
  select $E, r$ such that $C$
  $\{I \land C\}$ /* $C \Rightarrow \exists E, r (C)*/
  s := \text{conclusion}(s, r, E); /* updates state $s*/
  $\{I \land \text{later}_\text{state}(s, s')\}$ /* $\exists E, r (C)$ is undefined, $s'$ is earlier state */
end
$\{I \land \neg \exists E, r (C)\}$
```

In other words, arbitrarily apply any rule whose condition is true.

The foundation of the formalized schema are the rules defining the event operator semantics. To ensure that theorems hold, that proof methods are applicable, and that functions are useful, then the event operator semantics must meet the requirements of the rule forms addressed in the following paragraph. Basically, the event operators must follow the proposed structure to ensure significant parts of our formalized schema.

Event operator semantics can be defined by two categories of rules: generation rules and non-generation rules. The generation rule category is used to define situations when a composite event occurrence can be constructed. The non-generation rule category consists of help rules, in particular invalidation rules, that are used to define situations when event occurrences of

---

2The $C$ (a lexical substitution) is used in two different ways in this definition. Firstly, it is used in the while statement where $E$ and $r$ are properly bound. Secondly, it is used in the predicates between statements where $E$ and $r$ are bound to the variables declared in the algorithm.
contributing event types cannot be part of any future composite event occurrences. These rule categories are defined using two canonical rule forms: usage rule form and invalidation rule form. The usage rule form can be used to define both generation and non-generation rules, whereas the invalidation rule form can only be used to define non-generation rules. First, the usage rule form is defined followed by the invalidation rule form. To improve readability of the rules, the lexical substitution $X \cup Y = X^{\prime}$ is used.

Both rule forms are written $\forall E \forall \gamma_1, \ldots, \gamma_n \in G (\ldots \rightarrow C)$ where the bound variables $\gamma_1, \ldots, \gamma_n$ represents the candidate event occurrences (related to $E$) where at least one candidate is used in the conclusion $C$ of the rule.

**Definition 6.12**

**Usage rule form:**

Let $R$ be the range (or domain) predicate that ensures that the event occurrences in the candidate event occurrences are of specific event types, and $R_c$ be the context predicate that ensures that candidate event occurrences can be used (or invalidated) according to the event context; let $H$ be the hypothesis of the rule addressing the operator semantics (e.g., in Def:s 4.3 to 4.11 on pp. 47–49); let $C$ be the conclusion written as a sequence of assignments of sets ($X$) updated as $X \cup \Delta X$ (e.g., $G \cup \Delta G$); then the usage rule form is written as:

$$
\forall E \forall \gamma_1, \ldots, \gamma_n \in G :
\begin{array}{c}
R \\
R_c \\
H \\
C
\end{array}
$$

This is equivalent to $\forall E \forall \gamma_1, \ldots, \gamma_n \in G (R \land R_c \land H \rightarrow C)$.

This is read: “for all event expressions $E$ and for all candidate event occurrences $\gamma_1$ to $\gamma_n$ in $G$ for which $R$, $R_c$, and $H$ hold assert conclusion $C$”. An example of an assertion is the addition of a composite event occurrence to $G$.

A detailed explanation of how the semantics of an event operator should be structured in a usage rule is as follows. The predicate $R$ ensures that
the referenced event occurrences are of specific event types in the event expression; for example, \( R = \lambda \text{type}(\gamma_1) = E_1 \land \ldots \land \text{type}(\gamma_n) = E_n \) is a typical definition of the predicate \( R \). The context predicate \( R_c \) defines how an initiator event occurrence and a terminator event occurrence relate to previously used initiators and terminators as well as future occurrences. Since \( R_c \) depends on the context (e.g., recent, chronicle, continuous), it is parameterized by the initiator and terminator occurrence as well as a set of possible event types for initiators and terminators. For example, in recent event context (in Section 4.5.4 on p. 55), the predicate \( R_c \) must hold true for a pair of event occurrences where terminator occurrence should be the earliest available event occurrence that has not been processed, and the initiator occurrence should be the closest preceding event occurrence; N.B., the initiator and terminator event occurrence are considered to be of different event types in this example. The predicate \( H \) is derived from an event semantics, for example, in Section 4.4 on p. 46. An example of the predicate \( H \) is that the candidate event occurrences must be in a particular order (cf. the sequence operator Def. 4.4 on p. 47).

**Definition 6.13**

**Invalidation rule form:**

Using the same assumptions for the usage rule form and let \( H' \) be the invalidation hypothesis such that \( \forall E \forall \gamma_1, \ldots, \gamma_m \in \mathcal{G} (R \Rightarrow (\neg H' \Rightarrow H)) \); let \( BV IC = \lambda \{\gamma_i, \ldots, \gamma_n\} \cap \{\gamma_j, \ldots, \gamma_p\} = \emptyset \) be the bound variable interrelation constraint, and \( R'_c \) is either the part of the event context that is associated with the initiator or the terminator, then the invalidation rule form is written as:

\[
\forall E \forall \gamma_1, \ldots, \gamma_n \in \mathcal{G} : \\
\begin{align*}
R_{in} \\
R'_c \\
\forall \gamma_j, \ldots, \gamma_p \in \mathcal{G} (R_{jp} \land BV IC \land R_c \Rightarrow H')
\end{align*}
\]

\[\text{rule name}
\]

An equivalent form is \( \forall E \forall \gamma_1, \ldots, \gamma_n \in \mathcal{G} (R_{in} \land R'_c \forall \gamma_j, \ldots, \gamma_p \in \mathcal{G} (R_{jp} \land BV IC \land R_c \Rightarrow H') \rightarrow C) \).

□
The invalidation rule form is similar to the usage rule form. The major differences are: that the subpredicate $R$ is split into $R_{in}$ and $R_{jp}$, that the hypothesis $H'$ is a partial negation of $H$, and that the context predicate $R_c$ is used in two different places (partially in the subpredicate $R'_c$ and completely in the subpredicate $R_c$). The reason for the enclosed universal quantification in the invalidation rule is that there may be no occurrences of $(\gamma_j, \ldots, \gamma_p)$ in $G$ fulfilling $R_{jp}$ and, therefore, the $\gamma_i, \ldots, \gamma_n$ should be invalidated. This requires that the range predicate $R$ is split into $R_{in}$ and $R_{jp}$ addressing the different $\gamma$ variables (fulfilling $BVIC$). $H'$ is a partial negation of $H$ to define the situation where either a potential initiator event occurrence or a potential terminator event occurrence is invalidated. In an invalidation rule, $C$ is a set of assignments of invalidation sets.

In this work, the rules are named using the first letters of the operator, for example, the rule for sequence is named ‘s’ and chronicle aperiodic is named ‘ca’. Non-generation rules are primed, double-primed etc.

### 6.2.1 Basis for Constructing Event Operator Rules

In order to construct event operator rules, it is necessary to precisely define how the $condition(s, r, E)$ (in Def. 6.11 on p. 109) can be derived from the canonical rule forms. Moreover, a similar predicate, named $range(s, r, E)$ that is true if the range predicate part of a rule is true, is presented in the same way. Given these two predicates, it is then possible to state necessary requirements of operator rules that follow our schema. Additionally, the function $conclusion(s, r, E)$ has to be derived from the canonical rule forms.

**Definition 6.14**

**Apply conclusion function:**

The function $s = apply(C, s', E)$ reads “apply assignments in $C$ to state $s'$ for event type $E$ and return the new state $s$.

$\square$

In other words, the apply function is informally specified as the transformation from a state into another state.
Definition 6.15
Usage rule form predicates:
Let \( s \) be a state, and let \( r \triangleq \forall E \forall \gamma_1, \ldots, \gamma_n \in G (R \land R_c \land H \rightarrow C) \), then
\[
\text{condition}(s, r, E) \triangleq \ r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G \ (R \land R_c \land H)
\]
\[
\text{range}(s, r, E) \triangleq r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G \ (R \land R_c)
\]
\[
\text{conclusion}(s, r, E) \triangleq \text{apply}(C, s, E)
\]

Definition 6.16
Invalidation rule form predicates:
Let \( s \) be a state, and let \( r \triangleq \forall E \forall \gamma_1, \ldots, \gamma_n \in G (R_{in} \land R_c \land \forall \gamma_j, \ldots, \gamma_p \in G (R_{jp} \land BVIC \land R_c \rightarrow H') \rightarrow C) \), then
\[
\text{condition}(s, r, E) \triangleq r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G \ (R_{in} \land R_c \land \forall \gamma_j, \ldots, \gamma_p \in G \ (R_{jp} \land BVIC \land R_c \rightarrow H'))
\]
\[
\text{range}(s, r, E) \triangleq r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G \ (R_{in} \land R_c \land \forall \gamma_j, \ldots, \gamma_p \in G \ (R_{jp} \land BVIC \land R_c \rightarrow \text{true}))
\]
\[
\text{conclusion}(s, r, E) \triangleq \text{apply}(C, s, E)
\]

6.2.2 Event Expression Property Functions
To state properties of any event specification language defined in our schema, certain functions defined over event types for each event specification language are required. These functions return properties of event expressions, for example, it is desirable to state that recursive event type specifications are disallowed in a formal way for every event specification language defined in our schema; this requires a function that for any event type returns all contributing event types regardless of the actual event specification language. This function, all_operands(\( E \)), is informally defined as returning the set of immediate contributing event types of an event type. For example, for the sequence event operator in Solicitor: all_operands(\( E_1; E_2 \)) = \{ \( E_1, E_2 \) \}. Given this function, the formal way of stating that recursive event type specifications are disallowed is: \( \forall n, E_1, E_2, \ldots, E_{n-1}, E_n (n \geq 1 \land E_1 \in \text{all_operands}(E_2) \land E_2 \in \text{all_operands}(E_3) \land \ldots \land E_{n-1} \in \text{all_operands}(E_n) \Rightarrow E_n \notin \text{all_operands}(E_1)) \).
Definition 6.17

Event Expression Property Functions for an event specification language:
To be more specific, it is necessary to define functions for an event specification language that:

1. for the principal event operator of an event type
   (a) $\text{all\_operands}(E)$: return a set of all immediate contributing event types (e.g., $\{E_1, E_2\}$ are the immediate contributing event types of $E_1;E_2$ regardless of whether $E_1$ and $E_2$ are primitive or composite
   (b) $\alpha(E)$: return all immediate potential initiating contributing event types (e.g., the $E_1$ is an immediate potential initiating contributing event type of $E_1;E_2$
   (c) $\omega(E)$: return all immediate potential terminating contributing event types (e.g., the $E_2$ is an immediate potential terminating contributing event type of $E_1;E_2$

2. for the whole event expression defining an event type
   (a) $\hat{\text{all\_operands}}(E)$: return a set of all primitive contributing event types (e.g., $\{E_1, E_2, E_3\}$ is the set of all primitive contributing event types of $(E_1;E_2);E_3$)
   (b) $\hat{\alpha}(E)$: return a set of all primitive potential initiating contributing event types (e.g., $\{E_1\}$ is the set of all primitive potential initiating contributing event types of $(E_1;E_2);E_3$
   (c) $\hat{\omega}(E)$: return a set of all primitive potential terminating contributing event types (e.g., $\{E_3\}$ is the set of all primitive potential terminating contributing event types of $(E_1;E_2);E_3$

These functions are also sufficient for defining properties of an event specification language. This proposal relies on three facts:

1. The functions separate between contributing event types in terms of all, initiating, and terminating event types. This separation is sufficient, since non-initiating and non-terminating event types can be derived.
6.2 The Event Composition Schema

2. The functions cover all primitive contributing event types as well as the contributing event types of a principal event operator of an event type. The latter function can be used to define the former function.

3. Properties can be specified recursively over event expressions

6.2.3 Constructing Context Predicates

As mentioned earlier (in Section 6.2 on p. 102), the context predicate is used to select an initiator and a terminator occurrence with respect to used and invalidated occurrences. A central issue for context predicates is the earliest available occurrences of an event type, since a context predicate should be true for these occurrences out of all generated and processed event occurrences. Before giving a precise definition of the earliest available occurrence for an event context, consider the following example. Let $\mathcal{E}_\alpha$ be the set of event types that can initiate $E$, let $\mathcal{A}LL_\alpha = \{\gamma|\gamma \in \mathcal{G} \land \text{type}(\gamma) \in \mathcal{E}_\alpha\}$, let $\mathsf{USED}_\alpha = \text{iot}(\mathsf{U}_\alpha(E))$, let $\mathsf{INVAILEDETAED}_\alpha = \mathsf{I}_\alpha(E)$, let $\mathsf{CONSUMED}_\alpha = \mathsf{USED}_\alpha \cup \mathsf{INVAILEDETAED}_\alpha$, then the earliest available occurrences for expression $E$ in chronicle event context are $\mathcal{A}LL_\alpha \setminus \mathsf{CONSUMED}_\alpha$. In other words, the earliest available occurrences are generated but neither used nor invalidated. The earliest available event occurrences of an event type are generated by contributing event types, not invalidated, not expired (discarded with respect to a time constraint as described in Section 4.5.6 on p. 62), and reusable for several composite event occurrences if the event context allows this as, for example, in recent event context.

The earliest available event occurrences is a central issue, because it simplifies the proofs. The proof of correctness of rule evaluation can be divided into two cases: (i) a proof that the context predicate is true for the earliest available event occurrences, (ii) a proof that, given any permutation of temporal ordering of earliest available event occurrences, the correct rule is applied, and (iii) a proof concerning liveness (the requirement of enabling progress in operator rules is defined in Def. 6.11 on p. 109). Without this separation, a proof of correctness must be applied to all significant earliest available event occurrences; this may result in significantly more cases to prove.
6.2.4 Constructing Operator Rules

As mentioned earlier, an event operator is defined by a set of operator rules \( \{r, r', \ldots, r^{(n)}\} = \text{rules}(E) \) where \( E \) is an event type whose principal operator the rules define. A necessary constraint is defined in Lemma 6.1.

Lemma 6.1

Intra-event-type rule selection conflict avoidance:

\[
\forall E, s, r, r' (r, r' \in \text{rules}(E) \land r \neq r' \Rightarrow \\
\neg (\text{condition}(s, r, E) \land \text{condition}(s, r', E)))
\]

□

In other words, for all event types, for all possible states, for all possible rules associated with the principal operator of an event type, there must be exactly one rule associated to the operator for which the condition is true. This is necessary to avoid indeterminism (and thus conflicts) in the intra-event-type rule selection for composing an event occurrence.

Proof: Given that two rules associated with the principal operator of an event type can be true for the same state \( s \), then conflict resolution is necessary. This can complicate the proof, since additional properties with respect to conflict resolution are necessary to state and to prove. In the worst case, conflict resolution is arbitrary leading to indeterministic results. Q.E.D.

A sufficient constraint needs to ensure that all possible permutations of temporal ordering of event occurrences of contributing event types of an event type are part of all states. This is addressed in Lemma 6.3 on p. 118. To present this lemma, Def. 6.18 on the facing page is defined.
Definition 6.18
Valid temporal ordering of event occurrences:

\[\text{valid\_temporal\_permutation}(n, \gamma_1, \ldots, \gamma_n) \triangleq \]
\[\forall m \forall \Omega_1^{\downarrow 1}, \ldots, \Omega_n^{\downarrow 1}, \ldots, \Omega_{(n-1)}^{\downarrow 1}\]
\[\ldots \]
\[\forall \Omega_1^{\downarrow m}, \ldots, \Omega_n^{\downarrow m}, \ldots, \Omega_{(n-1)}^{\downarrow m}\]
\[\forall \mathcal{OP}'_R(\mathcal{OP}'_R \in \mathcal{P}\mathcal{OP}'_R \land m = |\mathcal{OP}'_R| \land \]
\[\Omega_1^{\downarrow 1}, \ldots, \Omega_1^{\downarrow m}, \ldots, \Omega_n^{\downarrow 1}, \ldots, \Omega_{(n-1)}^{\downarrow m}\]
\[\ldots \]
\[\forall \Omega_1^{\downarrow m}, \ldots, \Omega_n^{\downarrow m}, \ldots, \Omega_{(n-1)}^{\downarrow m} \in \mathcal{OP}'_R \land \Rightarrow \]
\[\forall i, k, l (1 \leq m \leq n \land 1 \leq k < l \leq n \Rightarrow \gamma_k \cap \gamma_i)\]
\[\square\]

Lemma 6.2
Minimum coverage of states:
Given the temporal ordering operators of event occurrences in \(\mathcal{OP}_R\), then the following must hold true (the function all_operands(\(E\)) is informally defined in Def. 6.17 on p. 114):

\[\forall n \forall E \forall E_1, \ldots, E_n \forall \gamma_1, \ldots, \gamma_n \forall s_i, \mathcal{G}_i, \mathcal{P}_i\]
\[(1 \leq n \leq |\text{all\_operands}(E)| \land E_1, \ldots, E_n \in \text{all\_operands}(E) \land \]
\[\forall k, l (1 \leq k < l \leq n \Rightarrow E_k \neq E_l) \land E_1 = \text{type}(\gamma_1) \land \ldots \land E_n = \text{type}(\gamma_n) \land \]
\[\text{valid\_temporal\_permutation}(n, \gamma_1, \ldots, \gamma_n) \land \]
\[s_i = \langle \mathcal{G}_i, \mathcal{P}_i \rangle \Rightarrow \]
\[\gamma_1, \ldots, \gamma_n \in \mathcal{G}_i\)\]

In other words, all possible combinations of event occurrences of contributing event types of each event type need to be part of \(\mathcal{G}\) to guarantee minimum coverage.
Proof: If this lemma does not hold, then some permutation is not covered and thus the proof is incomplete. N.B. for proofs of correctness (in practice), each permutation of temporal ordering of event occurrences needs to be tested once and the rest can be induced. Moreover, note that for event expressions whose contributing event types are primitive, this lemma represent the complete set of permutations that has to be checked. Q.E.D.

Lemma 6.3
Complete coverage of states:
To ensure complete coverage, given that Lemma 6.2 on the previous page holds, it is necessary to include repeated event occurrences of the contributing event types in such way that all possibilities of selection in the context predicates are evaluated.

Proof: If this lemma does not hold, then some permutation of repeated event occurrences is not covered and thus the proof is incomplete. Q.E.D.

Theorem 6.1
Equivalence check theorem: Given Lemmas 6.1 to 6.3 on pp. 116–118, let the predicate $\text{gen\_rule}(r)$ that is true for a rule $r$ that is a generation rule (otherwise it is false), and the following holds true:

$$\forall E, E' \forall r_1, \ldots, r_m, r'_1, \ldots, r'_n, s \quad (E \neq E' \land r_1, \ldots, r_m \in \text{rules}(E) \land r'_1, \ldots, r'_n \in \text{rules}(E') \land \text{gen\_rule}(r_1) \land \ldots \land \text{gen\_rule}(r_m) \land \text{gen\_rule}(r'_1) \land \ldots \land \text{gen\_rule}(r'_n) \Rightarrow (\text{condition}(s, r_1, E) \lor \ldots \lor \text{condition}(s, r_m, E) \Leftrightarrow \text{condition}(s, r'_1, E') \lor \ldots \lor \text{condition}(s, r'_n, E')))$$

In other words, for all states, for all generation rules of two different event types, the conditions of the generation rules need to be equivalent.

In this situation, it is only necessary to check the outcome for all permutations of temporal ordering of event occurrences of contributing event types in these generation rules to see if they are equivalent.
Proof: Axiom 6.2 on p. 98 guarantees that there are no ambiguities concerning what primitive event occurrences of the same event type to choose from in a specific situation. Without this axiom, it is impossible to check equivalence since the result can be random. Lemma 6.1 on p. 116 guarantees that no two rules for the same event type are true for the same state \( s \) and Lemma 6.3 on the facing page guarantees that all possible permutations of temporal orderings of event occurrences are covered. Since these permutations are covered, it is guaranteed that if the generation rules generate the same number of composite event occurrences with the same time spans, then they are equivalent.

Q.E.D.

Finally, a requirement of event operator rules is that they enable progress (as defined in Def. 6.11 on p. 109), where \( \text{later}_\text{state}(s, s') \) is required. This is defined in Theorem 6.2.

Theorem 6.2
Event operator rule progress:
All rules follow the usage and invalidation rule forms and in each rule at least one of the candidate event occurrences must be assigned to used or invalidated set of the associated event type to enable progress.

Proof: Given Axiom 6.1 on p. 97 it is never necessary to wait for the completion of another primitive event occurrence and thereby progress is enabled at the primitive event type level. If primitive event types would have intervals, then it may not be the termination time that determines the order of occurrence, but the initiation time. If the initiation time determines the order, then it may be the case that a primitive event occurrence must be unnecessarily or infinitely delayed.

If a rule would not follow neither the usage nor the invalidation rule form, then progress can be disabled since candidate event occurrences are not checked with the context predicate. Additionally, if a rule following either the usage rule form or the invalidation rule form is allowed to do nothing, then it can be repeatedly applied without making progress. Thus, such rules must be disallowed to enable progress.

Q.E.D.
6.3 The Solicitor Event Specification Language

First, basic assumptions concerning event types are addressed followed by expiration semantics and context predicates. Finally, operator rules are defined and proved to be correct.

6.3.1 Basic Assumptions Concerning Event Types

To handle conflicts in event composition, it is assumed that event types are ordered (cf. Section 4.6 on p. 63). A conflict can occur when two occurrences of different primitive event types terminate simultaneously. For example, $E = E_1; (E_2 \lor E_3)$ can give rise to conflicts since occurrences of $E_2$ and $E_3$ may terminate simultaneously. In chronicle context, given an occurrence of $E_1$, then what occurrence of either $E_2$ or $E_3$ should be used (or both if the semantics of the disjunction is inclusive)?

Ordering for arbitration has been applied in distributed systems to resolve temporal order conflicts. This is an efficient method to resolve conflicts of this kind; for example, distributed algorithms use a unique site identifier to resolve conflicts (e.g., [GM82]). Further, it is assumed that event occurrences are sent via stable messages [Sch94a]. That is, the event occurrences are not delivered until it is guaranteed that all messages arriving at time $t$ have been received.

Definition 6.19

Conflict order arbitration of event types:
$\text{c\text{-ord}}(E)$ returns the order of event type $E$, where $\text{c\text{-ord}}(E) \leq \text{c\text{-ord}}(E')$ iff event type $E$ is of lower or equal order compared to event type $E'$.

□

The event composition must process event types in order to avoid conflicts, for example, if $\text{c\text{-ord}}(E_2) < \text{c\text{-ord}}(E_3)$ then all event occurrences of $E_2$ precedes simultaneous event occurrences of $E_3$. It is necessary to assume that Def. 6.11 on p. 109 processes rules to maintain conflict order relations. The definition of a control-driven scheduling of event composition maintaining conflict order relations is elaborated in Ch. 7. N.B. any solution using a centralized agent, such as semi-active replication [KV94, p. 435], determines the order in which messages (delivering event occurrences) should be
6.3 The Solicitor Event Specification Language

processed can employ the results in this work; the reason is that event occurrences can receive discriminating timestamps by the centralized agent that can be viewed as temporary c_ord relations; these temporary c_ord relations hold for the duration of event composition.

To compare event occurrences in terms of temporal order, it is often necessary to consider the contributing event occurrences. In particular, the initiator and terminator occurrences of a composite occurrence are of interest. Given \( E = E_1; E_4 \) where \( E_4 = (E_2 \lor E_3) \) and \( \mathcal{G} = \{ \Gamma(E_2, [0,1]), \Gamma(E_3, [1,1]), \Gamma(E_4, [0,1]) \} \), then it is not possible to discriminate between the occurrences \( \Gamma(E_4, [0,1]) \) and \( \Gamma(E_4, [1,1]) \) when composing an event occurrence of \( E \) without considering the terminating constituent \( \Gamma(E_2, [0,1]) \) and \( \Gamma(E_3, [1,1]) \). To enable comparison of contributing event occurrences, we introduce Def:s 6.20 to 6.23 on pp. 121–122 whose uniqueness of the result relies on Axiom 6.3. This axiom is based on predicates defined in Def:s 6.3 to 6.4 on p. 105.

**Axiom 6.3**
Unique contributing event occurrences of composite event occurrences:

\[
\forall E \forall \gamma, \gamma', \gamma'' \in \mathcal{G} \ (\text{type}(\gamma) = E \land \text{initiate}(\gamma', \gamma) \land \text{terminate}(\gamma'', \gamma)) \Rightarrow \\
\forall \gamma''' \in \mathcal{G} \ (\gamma' \neq \gamma''' \land \gamma'' \neq \gamma''') \Rightarrow \\
\neg \text{initiate}(\gamma''', \gamma) \land \neg \text{terminate}(\gamma''', \gamma))
\]

□

In other words, for each composite event occurrence, there is at the most one initiator and terminator event occurrence (possibly the same event occurrence).

**Definition 6.20**
Immediate terminator occurrence: Let \( \gamma \overset{\Delta}{=} \Gamma(E, [t, t']) \in \mathcal{G} \), then

\[
\omega \uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff prim}(E) \\
\gamma' \text{where } \gamma' \in \{ \gamma'' | \text{terminate}(\gamma'', \gamma) \} & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

□
In other words, return the event occurrence that terminates \( \gamma \). Given Axiom 6.3 on the previous page, the result is unique since the cardinality of \( \{ \gamma'' | (\gamma'', \gamma) \in U_\omega(E) \} \) is one or zero.

**Definition 6.21**

**Ultimate terminator occurrence:** Let \( \gamma \Delta \Gamma(E, [t, t']) \in \mathcal{G} \), then

\[
\hat{\omega} \uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff } \text{prim}(E) \\
\hat{\omega} \uparrow(\omega \uparrow(\gamma)) & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

\( \square \)

In other words, \( \hat{\omega} \uparrow(\gamma) \) returns the ultimate (primitive) event occurrence that terminated \( \gamma \).

**Definition 6.22**

**Immediate initiator occurrence:** Let \( \gamma \Delta \Gamma(E, [t, t']) \in \mathcal{G} \), then

\[
\alpha \uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff } \text{prim}(E) \\
\gamma' \text{ where } \gamma' \in \{ \gamma'' | \text{initiate}(\gamma'', \gamma) \} & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

\( \square \)

**Definition 6.23**

**Ultimate initiator occurrence:** Let \( \gamma \Delta \Gamma(E, [t, t']) \in \mathcal{G} \), then

\[
\hat{\alpha} \uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff } \text{prim}(E) \\
\hat{\alpha} \uparrow(\alpha \uparrow(\gamma)) & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

\( \square \)

Defs 6.22 to 6.23 on this page are similar to Defs 6.20 to 6.21 on pp. 121–122, but return the event occurrence that initiates \( \gamma \).
Then, the following temporal ordering operators on event occurrences are defined:

Definition 6.24
Precedes:

\[ \gamma_1 \prec \gamma_2 \triangleq \text{end}(\text{span}(\gamma_1)) < \text{end}(\text{span}(\gamma_2)) \lor \]
\[ \text{end}(\text{span}(\gamma_1)) = \text{end}(\text{span}(\gamma_2)) \land \]
\[ ((\text{c}_\text{ord}(\text{type}(\gamma_1)) < \text{c}_\text{ord}(\text{type}(\gamma_2)) \lor \omega(\gamma_1) \prec \omega(\gamma_2))) \]

The ‘precedes’ operator is irreflexive, transitive, and asymmetric. That is, \( \forall \gamma \, (\neg (\gamma \prec \gamma)) \), \( \forall \gamma, \gamma', \gamma'' \, (\gamma \prec \gamma' \land \gamma' \prec \gamma'' \Rightarrow \gamma \prec \gamma'') \), and \( \forall \gamma, \gamma' \, (\gamma \prec \gamma' \Rightarrow \neg (\gamma' \prec \gamma)) \).

Definition 6.25
Succeeds:

\[ \gamma_1 \succ \gamma_2 \triangleq \gamma_2 \prec \gamma_1 \]

Similarly to the ‘precedes’ operator, the ‘succeeds’ operator is irreflexive, transitive, and asymmetric.

Definition 6.26
Overlaps:

\[ \gamma_1 \parallel \gamma_2 \triangleq \neg (\text{end}(\text{span}(\gamma_1)) < \text{start}(\text{span}(\gamma_2)) \lor \text{start}(\text{span}(\gamma_1)) > \text{end}(\text{span}(\gamma_2))) \]

\( \Box \)
The overlaps operator is reflexive, non-transitive, and symmetric. That is, \( \forall \gamma (\gamma \parallel \gamma) \), \( \neg \forall \gamma, \gamma', \gamma'' (\gamma \parallel \gamma' \land \gamma' \parallel \gamma'' \Rightarrow \gamma \parallel \gamma'' \)\), and \( \forall \gamma, \gamma' (\gamma \parallel \gamma' \Rightarrow \gamma' \parallel \gamma) \).

The 'precedes' relation is based on the termination time of event occurrences, since this determines the time when subscribers of an event type are notified of its occurrence.

6.3.2 Expiration Semantics

As discussed in Section 4.5.6 on p. 62, expiration (or validity interval) is an important feature to limit processing as suggested by Dousson et al. [DGG93, Dou96] and to detect violation of time constraints early as suggested by Mok et al. [ML97a]. In this chapter, specification of expiration is introduced in Solicitor.

As mentioned in Section 4.3.1 on p. 43, the \( \text{expire} \) attribute is used to specify the expiration time. For example, \( E_1; (\text{expire}: 10s) E_2 \) means that the maximum duration of an \( E_1 \) occurrence followed by an \( E_2 \) occurrence is 10 seconds. By default, \( \forall E,E' (E' \in \text{operands}(E) \Rightarrow \text{expire}(E) = \text{expire}(E')) \).

**Definition 6.27**

Expiration predicate:

\[
\text{expired}(\gamma) \triangleq \text{now} - \text{start}(\text{span}(\gamma)) > \text{expire}(\text{type}(\gamma))
\]

\( \Box \)

In other words, \( \text{expired}(\gamma) \) is true if \( \gamma \) has expired. N.B. it is assumed that event composition is invoked so that no event occurrence is incorrectly expired due to late invocation.

6.3.3 Event Expression Property Functions in Solicitor

The necessary functions in Def. 6.17 on p. 114 for Solicitor are defined in Defs 6.30 to 6.31 on pp. 125–126 and Defs 6.32 to 6.33 on p. 126. These are used to derive event types whose occurrences can initiate and terminate a composite event respectively. For example, \( \hat{\alpha}(E_1 \triangle (E_2; E_3)) = \{ E_1, E_2 \}, \) \( \alpha(E_1 \triangle (E_2; E_3)) = \{ E_1, E_2; E_3 \}, \) \( \hat{\omega}(E_1 \triangle (E_2; E_3)) = \{ E_1, E_3 \}, \) and \( \omega(E_1 \triangle \)
\( (E_2; E_3) = \{ E_1, E_2; E_3 \} \). N.B., the chronicle aperiodic event operator (i.e., \( CA(E_1, E_2, E_3) \)), addressed in these functions, is defined in Ch. 8.

**Definition 6.28**

All-immediate-contributing-types-of:

\[
\text{all\_operands}(E) = \begin{cases} 
(a): \{ E \} & \text{iff } \text{prim}(E) \\
(b): \{ E_1, E_2 \} & \text{iff } (E = E_1; E_2) \lor (E = E_1 \nabla E_2) \lor (E = E_1 \triangle E_2) \\
(c): \{ E_1, E_T \} & \text{iff } (E = E_1 + T) \\
(d): \{ E_1, E_2, E_3 \} & \text{iff } (E = N(E_1, E_3, E_2)) \lor (E = A(E_1, E_2, E_3)) \lor (E = CA(E_1, E_2, E_3)) 
\end{cases}
\]

\[ \square \]

**Definition 6.29**

All-contributing-types-of:

\[
\text{all\_operands}(E) = \begin{cases} 
(a): \{ E \} & \text{iff } \text{prim}(E) \\
(b): \bigcup_{E' \in \text{all\_operands}(E)} \text{all\_operands}(E') & \text{iff } \neg \text{prim}(E) 
\end{cases}
\]

\[ \square \]

**Definition 6.30**

Immediate-contributing-initiator-types-of:

\[
\alpha(E) = \begin{cases} 
(a): \{ E \} & \text{iff } \text{prim}(E) \\
(b): \{ E_1 \} & \text{iff } (E = E_1; E_2) \lor (E = N(E_1, E_E), E_2)) \lor (E = E_1 + T) \\
(c): \{ E_1, E_2 \} & \text{iff } (E = E_1 \nabla E_2) \lor (E = E_1 \triangle E_2) \\
(d): \{ E_2 \} & \text{iff } (E = A(E_1, E_2, E_3) \lor E = CA(E_1, E_2, E_3) 
\end{cases}
\]

\[ \square \]
Definition 6.31
Contributing-initiator-types-of:

\[\hat{\alpha}(E) = \begin{cases} 
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \bigcup_{E' \in \alpha(E)} \hat{\alpha}(E') & \text{iff } \neg \text{prim}(E) 
\end{cases} \]

\[\square\]

Definition 6.32
Immediate-contributing-terminator-types-of:

\[\omega(E) = \begin{cases} 
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \{E_2\} & \text{iff } (E = E_1; E_2) \lor (E = N(E_1, E', E_2)) \\
(c): \{E_1, E_2\} & \text{iff } (E = E_1 \lor E_2) \lor (E = E_1 \Delta E_2) \\
(d): \{E_2\} & \text{iff } E = A(E_1, E_2, E_3) \lor \\
& \quad E = CA(E_1, E_2, E_3) 
\end{cases} \]

\[\square\]

Definition 6.33
Contributing-terminator-types-of:

\[\hat{\omega}(E) = \begin{cases} 
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \bigcup_{E' \in \omega(E)} \hat{\omega}(E') & \text{iff } \neg \text{prim}(E) 
\end{cases} \]

\[\square\]

6.3.4 Context Predicates in Solicitor

In this section, we define the event context predicates for chronicle, recent, continuous, and general event context (cf. Section 4.5 on p. 51) as a proof of concept of our formalized schema, since these event contexts are considered to be the complete set of meaningful event contexts. Our formalization shows that these are indeed the complete set of basic meaningful event contexts (cf. Table 7.1 on p. 165). Our presentation is an extension of the formalism in Section 4.5 on p. 51 that starts with general definitions and notation, followed by the event contexts where chronicle is presented first. The reason
why it is presented first is that recent and continuous event context are based on the same subpredicates as the chronicle event context.

The context predicate is underlined (e.g., “recent”). There is a context predicate for each context (e.g., recent, chronicle). These predicates have five parameters. These parameters are the currently evaluated event type \( (E) \), the current potential initiator occurrence \( (\gamma_\alpha) \), the current potential terminator \( (\gamma_\omega) \), a set of event types whose occurrences may be initiators \( (E_\alpha) \), and a set of event types whose occurrences may be terminators \( (E_\omega) \). \( E_\alpha \) and \( E_\omega \) are necessary, because an operator rule may consider only a subset of \( \alpha(E) \) or \( \omega(E) \); this is discussed in detail in Section 6.3.5 on p. 134.

Each context predicate consists of three subpredicates as defined in Def. 6.34: the initiator predicate (subscripted with ‘\( \alpha \)’), the terminator predicate (subscripted with ‘\( \omega \)’), and the initiator-terminator predicate (subscripted with ‘\( \alpha\omega \)’). The first two subpredicates take three parameters: \( E, \gamma_\alpha \) (or \( \gamma_\omega \)), and \( E_\alpha \) (or \( E_\omega \)). The last subpredicate takes the same 5 parameters as the context predicate. The main reason for this division of the context predicate into subpredicates is mainly pedagogical.

**Definition 6.34**

**General form of context predicate:**

Let \( X \) be the context predicate, then:

\[
X(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \overset{\Delta}{=} \ X_\alpha(E, \gamma_\alpha, E_\alpha) \land X_\omega(E, \gamma_\omega, E_\omega) \land X_{\alpha\omega}(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega)
\]

\[ \square \]

For example, the context predicate for the chronicle event context is written:

\[
\text{chronicle}(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \overset{\Delta}{=} \ \text{chronicle}_\alpha(E, \gamma_\alpha, E_\alpha) \land \ \text{chronicle}_\omega(E, \gamma_\omega, E_\omega) \land \ \text{chronicle}_{\alpha\omega}(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega)
\]

To give the precise definitions of the event context predicates, Defs 6.35 to 6.38 on the following page defines applicable functions and dynamic sets.
Definition 6.35

All Non-expired Event Occurrences of a Set of Event Types:

\[
\text{all	extunderscore of	extunderscore types}(\mathcal{E}) \triangleq \{ \gamma | \gamma \in \mathcal{G} \land \text{type}(\gamma) \in \mathcal{E} \land \neg \text{expired}(\gamma) \}
\]

□

Definition 6.36

Basic dynamic set definitions for context predicates:

\[\text{ALL}^\lambda_\alpha = \text{all	extunderscore of	extunderscore types}(\mathcal{E}_\alpha)\] (all event occurrences of initiator event types \(\mathcal{E}_\alpha\))

\[\text{ALL}^\lambda_\omega = \text{all	extunderscore of	extunderscore types}(\mathcal{E}_\omega)\] (all event occurrences of terminator event types \(\mathcal{E}_\omega\))

□

Definition 6.37

Dynamic set definitions of event type \(E\) for context predicates:

\[\text{COMP}^\lambda = (\text{com}(\mathcal{U}_\alpha(E) \cup \mathcal{U}_\omega(E)))\] (all composed event occurrences)

\[\text{INVAL}_\alpha^\lambda = \text{INVAL}(\mathcal{E}_\alpha)\] (invalid as initiators to form composite event occurrences)

\[\text{INVAL}_\omega^\lambda = \text{INVAL}(\mathcal{E}_\omega)\] (invalid as terminators to form composite event occurrences)

\[\text{USED}_\alpha^\lambda = \text{iot}(\mathcal{U}_\alpha(E))\] (used as initiators for composite event occurrences)

\[\text{USED}_\omega^\lambda = \text{iot}(\mathcal{U}_\omega(E))\] (used as terminators for composite event occurrences)

□

Definition 6.38

Derived dynamic set definitions of event type \(E\) for context predicates:

\[\text{CONS}_\alpha^\lambda = (\text{USED}_\alpha \cup \text{INVAL}_\alpha)\] (consumed as initiators)

\[\text{CONS}_\omega^\lambda = (\text{USED}_\omega \cup \text{INVAL}_\omega)\] (consumed as terminators)

\[\text{UNCONS}_\alpha^\lambda = (\text{ALL}_\alpha \setminus (\text{CONS}_\alpha \cup \text{CONS}_\omega))\] (unconsumed initiators)

\[\text{UNCONS}_\omega^\lambda = (\text{ALL}_\omega \setminus (\text{CONS}_\alpha \cup \text{CONS}_\omega))\] (unconsumed terminators)

\[\text{VALID}_\alpha^\lambda = (\text{ALL}_\alpha \setminus \text{INVAL}_\alpha)\] (initiators valid for composition)

\[\text{VALID}_\omega^\lambda = (\text{ALL}_\omega \setminus \text{INVAL}_\omega)\] (terminators valid for composition)

□

Given these definitions, the precise definition of chronicle event context is in Def. 6.39 on the facing page.
Definition 6.39
Chronicle context predicate:

\[
\text{chronicle}_\alpha(E, \gamma_\alpha, E_\alpha) \triangleq \forall \gamma \in UNCONS_\alpha (\gamma_\alpha \neq \gamma \Rightarrow \gamma_\alpha \prec \gamma) \land \gamma_\alpha \in UNCONS_\alpha
\]

\[
\text{chronicle}_\omega(E, \gamma_\omega, E_\omega) \triangleq \forall \gamma \in UNCONS_\omega (\gamma_\omega \neq \gamma \Rightarrow \gamma_\omega \prec \gamma) \land \gamma_\omega \in UNCONS_\omega
\]

\[
\text{chronicle}_\alpha\omega(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \triangleq true
\]

In other words, the initiator occurrence should be the earliest unconsumed initiator occurrence. The same applies to the terminator occurrence.

One thing that may seem odd is that the definition of \( UNCONS_\alpha \) (and \( UNCONS_\omega \)) removes both initiator and terminator occurrences \( CONS_\alpha \) (and \( CONS_\omega \)) from \( ALL_\alpha \) (and \( ALL_\omega \)). The reason is that some event operators may use the contributing event type either as a initiator or terminator depending on the event history (e.g., the conjunction operator). For such event operators, it is necessary to remove all elements from \( ALL_\alpha \) (and \( ALL_\omega \)) that have been used or invalidated either as terminator or as initiators.

Definition 6.40
Recent context predicate:

\[
\text{recent}_\alpha(E, \gamma_\alpha, E_\alpha) \triangleq \gamma_\alpha \in V\text{ALID}_\alpha
\]

\[
\text{recent}_\omega(E, \gamma_\omega, E_\omega) \triangleq \text{chronicle}_\omega(E, \gamma_\omega, E_\omega)
\]

\[
\text{recent}_\alpha\omega(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \triangleq \neg \exists \gamma \in V\text{ALID}_\alpha (\gamma_\alpha \neq \gamma \land \gamma_\alpha \prec \gamma \prec \gamma_\omega)
\]

In other words, there may be no initiator occurrence between the currently evaluated initiator and terminator occurrences. Terminators follow the same rule as for chronicle. In this context, some initiators are explicitly invalidated, whereas other initiators are skipped (or implicitly invalidated) since they cannot be selected by the context predicate.
Definition 6.41

Continuous context predicate:

\[
\text{continuous}_\alpha(E, \gamma_\alpha, E_\alpha) \triangleq \text{chronicle}_\alpha(E, \gamma_\alpha, E_\alpha)
\]

\[
\text{continuous}_\omega(E, \gamma_\omega, E_\omega) \triangleq \gamma_\omega \in \text{V}_{\text{ALID}_\omega}
\]

\[
\text{continuous}_{\alpha\omega}(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \triangleq \neg \exists \gamma \in \text{V}_{\text{ALID}_\omega}(\gamma_\omega \neq \gamma \land \gamma_\alpha \prec \gamma \prec \gamma_\omega)
\]

□

In other words, as for chronicle context, the initiator occurrence should be the earliest unconsumed initiator occurrence. For the terminator, there may be no later terminator occurrence used than the one we use for composition. Essentially, terminator occurrences can be reused for more than one composition.

Definition 6.42

General event context:

\[
\text{gen\textunderscore combination\textunderscore not\textunderscore used\textunderscore before}(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \triangleq
\forall \gamma, \gamma', \gamma'' (\text{type}(\gamma) = E \land \text{initiate}(\gamma', \gamma) \land \text{terminate}(\gamma'', \gamma) \land \gamma' \neq \gamma'' \Rightarrow
\neg (\gamma_\alpha = \gamma' \land \gamma_\omega = \gamma'' \land \neg (\gamma_\alpha = \gamma'' \land \gamma_\omega = \gamma')))
\]

\[
\text{all\textunderscore prev\textunderscore occs\textunderscore processed}_1(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \triangleq
\forall \gamma, \gamma' (\text{type}(\gamma) = E \land \text{initiate}(\gamma', \gamma) \land \gamma_\alpha \succ \gamma' \Rightarrow
\forall \gamma'' (\gamma'' \in \text{V}_{\text{ALID}_\omega} \land (\gamma' < \gamma'' \lor \gamma' = \gamma'') \Rightarrow
\exists \gamma''' (\text{type}(\gamma''') = E \land
\text{initiate}(\gamma', \gamma''') \land \text{terminate}(\gamma'', \gamma'''))) \land
\forall \gamma (\gamma \in \text{V}_{\text{ALID}_\omega} \land (\gamma_\alpha < \gamma \lor \gamma_\alpha = \gamma) \land \gamma_\omega > \gamma \Rightarrow
\exists \gamma' (\text{terminate}(\gamma', \gamma)))
\]

\[
\text{all\textunderscore prev\textunderscore occs\textunderscore processed}_2(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \triangleq
\forall \gamma (\gamma \in \text{V}_{\text{ALID}_\omega} \land \gamma_\alpha \succ \gamma \Rightarrow \exists \gamma' (\text{initiate}(\gamma, \gamma'))) \land
\forall \gamma (\gamma \in \text{V}_{\text{ALID}_\omega} \land \gamma_\omega > \gamma \Rightarrow \exists \gamma' (\text{terminate}(\gamma, \gamma')))
\]

\[
\text{general}_\alpha(E, \gamma_\alpha, E_\alpha) \triangleq \gamma_\alpha \in \text{V}_{\text{ALID}_\alpha}
\]
6.3 The Solicitor Event Specification Language

In other words, general event context is true for any combination of initiator or terminator that has not already contributed to a specific composite event occurrence. Further, the context predicate must enable liveness by guaranteeing progress; there are two cases: (i) the event type sets are different for event operators that generate composite event occurrences from two event occurrences of different contributing event types (e.g., the sequence operator), and (ii) the event type sets are equal for event operators that generate composite event occurrences from one contributing event type and treats it as both initiator and terminator (e.g., disjunction operator). In case (i) \((\text{all\_pre\_occ\_processed}_1)\) predicate, no initiator can be chosen without ensuring that preceding initiators have been completely matched with all valid terminators. Moreover, the earliest available valid terminator should be chosen for the currently evaluated initiator. N.B. for the conjunction event operator there is no static assignment in the rules of which contributing event types are initiators and terminators (cf. Def. 6.45 on p. 138); therefore, \(\gamma_\alpha\) must be compared to both \(\gamma'\) and \(\gamma''\) (and similarly for \(\gamma_\omega\)). In case (ii) \((\text{all\_pre\_occ\_processed}_2)\) predicate, the only requirement is that preceding valid event occurrences must have been used.

The reason for this complexity is that \(\text{general}_{\omega}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \overset{\triangleq}{=} \gamma_\omega \in \mathcal{VALID}_\omega\)

\[
\text{general}_{\omega}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \overset{\triangleq}{=}
\text{gen\_combination\_not\_used\_before}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \land
(E_\alpha \neq E_\omega \Rightarrow \text{all\_pre\_occ\_processed}_1(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega)) \land
(E_\alpha = E_\omega \Rightarrow \text{all\_pre\_occ\_processed}_2(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega))
\]

Non-overlapping contexts: To define non-overlapping contexts (cf. Section 4.5.5 on p. 61), \(\forall \gamma \in \mathcal{COMP} (\gamma \prec \gamma_\alpha \land \neg(\gamma \parallel \gamma_\alpha))\) is added as a conjunction.
in the $X_{\alpha\omega}$ part of the context predicate. For example, the context predicate for non-overlapping chronicle context is defined in Def. 6.43.

**Definition 6.43**

Non-overlapping chronicle context predicate:

\[
\text{n.o.chronicle}_\alpha (E, \gamma_\alpha, E_\alpha) \Delta = \text{chronicle}_\alpha (E, \gamma_\alpha, E_\alpha)
\]

\[
\text{n.o.chronicle}_\omega (E, \gamma_\omega, E_\omega) \Delta = \text{chronicle}_\omega (E, \gamma_\omega, E_\omega)
\]

\[
\text{n.o.chronicle}_{\alpha\omega} (E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) \Delta = \text{true} \land \forall \gamma \in \text{COMP} (\gamma < \gamma_\alpha \land \neg (\gamma \parallel \gamma_\alpha))
\]

**Brief comparison of event contexts:** As can be seen in the basic context predicates of Solicitor, there are similarities between them. For example, the selection of a terminator occurrence ($\gamma_\omega$) are the same for both the recent and chronicle event context. Also, the selection of initiator occurrence in chronicle event context is the same as in continuous event context.

**Lemma 6.4**

**Progress in Solicitor:**

Given that each rule assigns candidate event occurrences to used or invalidated sets during processing fulfilling Theorem. 6.2 on p. 119, then there will be progress since these assignments change the state of event composition.

**Proof:** First of all, all operator rules are required to apply the context predicate or relevant part thereof to candidate event occurrences. Given this, it is necessary to argue that all event contexts enable progress.

There is a case for each overlapping event context and one case for all non-overlapping event contexts: (i) chronicle context, (ii) recent context, (iii) continuous context, (iv) general context, and (v) non-overlapping variants. In case (i), when a candidate contributing event occurrence is used (to form a composite event occurrence of the contributed event type) it should be
added to the used set of initiators or terminators (in Def. 6.37 on p. 128). This leads to it being consumed (as initiator or terminator) (in Def. 6.38 on p. 128). The unconsumed event occurrences are defined in Def. 6.37 on p. 128 and are the basis for the chronicle context predicate (in Def. 6.39 on p. 129). Thus, using candidate event occurrences enables progress in the chronicle event context.

If, on the other hand, a candidate contributing event occurrence cannot be used to form a composite event occurrence, then it should be added to the invalidated set (as initiator or terminator). These are also part of the consumed event occurrences in Def. 6.38 on p. 128. Thus, invalidation of event occurrences enables progress in the chronicle event context.

To conclude case (i), as long as each operator rule whose precedent is true either use or invalidate at least one candidate event occurrence that is either an initiator or a terminator, then progress is enabled for chronicle event context.

Concerning case (ii) and (iii) whose context predicates share subpredicates with the chronicle context predicate, the aforementioned reasoning for chronicle event context holds for the shared subpredicates. To recapitulate the subpredicates from Def. 6.40 on p. 129 and Def. 6.41 on p. 130, recent\(_\omega\)(E,γ\(_\omega\),E\(_\omega\)) \(\triangleq\) chronicle\(_\omega\)(E,γ\(_\omega\),E\(_\omega\)) and continuous\(_\alpha\)(E,γ\(_\alpha\),E\(_\alpha\)) \(\triangleq\) chronicle\(_\alpha\)(E,γ\(_\alpha\),E\(_\alpha\)). Thus, progress is enabled for terminator occurrences in recent event context and initiator occurrences in continuous event context.

Concerning the initiators in case (ii) they must be the most recent occurrence of the valid event occurrences of the contributing event type preceding the currently evaluated terminator occurrence (in Def. 6.38 on p. 128). Given that the operator rules invalidate candidate event occurrences that cannot form a composite event occurrence of the contributed event type, then the recent context predicate will only be true for valid event occurrences. This enables progress in terms of invalidated event occurrences (initiators). Further, since the initiator and terminator are coupled, then enabled progress for the terminator implies enabled progress for the initiators and vice versa.

Concerning case (iii), the reasoning for terminators is the same as for initiators in recent event context.

Concerning case (iv), only valid occurrences are selected. So if the operator rules invalidate candidate contributing event occurrences that cannot be part of any future composite event occurrence of the contributed event type, then progress is enabled with respect to invalid event occurrences. Further, since Axiom 6.3 on p. 121 is required it is guaranteed that a pair of initiator
and terminator event occurrences are only used to form one composite event occurrence. This fact enables progress, since the general event context predicate does not hold true for any pair that has been used already.

Concerning case \((v)\), all event operators in all overlapping event contexts generate composite event occurrences in order; a necessary prerequisite for non-overlapping event context definitions, since non-overlapping definitions check which composite event occurrences has been generated to determine what can be composed. This prerequisite enables progress. \(\text{Q.E.D.}\)

### 6.3.5 Operator Rules

The event operators sequence, conjunction, and disjunctions are used as examples. The sequence operator is used to point out the basic operator rule constructs, whereas the conjunction and disjunction event operators are used to illuminate the reason for the design of the event context predicates (in Section 6.3.4 on p. 126). The rest of the event operators are defined in Appendix B.

In the rules of the sequence operator, let \(X\) represent the context predicate of the event type \((X = \text{context}(E))\). The operator rules for sequence are as follows:

**Sequence Operator**

**Definition 6.44**  
**Sequence:**

Let \(E = E_1; E_2\) and \(\gamma \equiv \Gamma(E, \lbrack \text{start} (\text{span}(\gamma_1)), \text{end} (\text{span}(\gamma_2)) \rbrack)\) in the \(s\)-rule:

\[
\forall E \forall \gamma_1, \gamma_2 \in \mathcal{G} : \\
\text{type}(\gamma_1) = E_1 \land \text{type}(\gamma_2) = E_2 \\
X(E, \gamma_1, \gamma_2, \{E_1\}, \{E_2\}) \\
\gamma_1 \prec \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2) \\
\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_s(E) \cup = \{\langle \gamma_1, \gamma \rangle\} \quad \mathcal{U}_w(E) \cup = \{\langle \gamma_2, \gamma \rangle\} \\
\]

Q.E.D.
∀E ∀γ₂ ∈ G :

\[ \text{type}(\gamma_2) = \text{E}_2 \]

\[ X_\omega(E, \gamma_2, \{ \text{E}_2 \}) \]

∀γ₁ ∈ G (type(γ₁) = \text{E}_1 \land X(E, \gamma_1, \gamma_2, \{ \text{E}_1 \}, \{ \text{E}_2 \}) \Rightarrow \gamma_1 \succ \gamma_2 \lor \gamma_1 \parallel \gamma_2) \quad s'\quad \text{I}_\omega(E) \cup = \{ \gamma_2 \}

□

In other words for the \( s \)-rule, for each pair of contributing event occurrences (γ₁ and γ₂) of each \( \text{E}_1; \text{E}_2 \) for which the context predicate \( X \) holds as well as the hypothesis (the \( \text{E}_1 \) occurrence precedes the \( \text{E}_2 \) occurrence and they do not overlap), event composition should do the following:

• assert that the composite event \( \text{E}_1; \text{E}_2 \) has occurred throughout the span \( \text{start}(\text{span}(\gamma_1)) \) and \( \text{end}(\text{span}(\gamma_2)) \) by adding the constructed event occurrence to \( G \)

• assert that \( \gamma_1 \) and \( \gamma_2 \) has been used as initiator and terminator occurrences (respectively)

In other words, for the \( s' \)-rule: if all \( \text{E}_1 \) occurrences that fulfill the context predicate either do not precede the \( \text{E}_2 \) occurrence or overlaps with the \( \text{E}_2 \) occurrence, then the \( \text{E}_2 \) occurrence is invalidated. The reason is that the currently evaluated terminator (if any) cannot be used as a terminator for any \( \text{E}_1 \) occurrence.

**Proof of Rule Correctness for Sequence Operator**

This is basically a proof of program safety (i.e., nothing bad will ever happen). Let \( \gamma_1 \) represent the earliest available \( \text{E}_1 \) occurrence, and \( \gamma_2 \) the earliest available \( \text{E}_2 \) occurrence. All possible permutations of temporal ordering as well as overlapping of earliest available event occurrences of \( \gamma_1 \) and \( \gamma_2 \) are considered in this proof. Since all permutations are considered (cf. Lemmas 6.1 to 6.3 on pp. 116–118), then all situations are covered.

The rules that are applicable to each of these permutations are summarized in Table 6.1 on the following page. A short form for the precedence
is used where the substitution rule is that \((i, \ldots, j) \overset{\lambda}{\rightarrow} \gamma_i \prec \gamma_{i+1} \prec \cdots \prec \gamma_j\). For example, \((1, 2)\) means \(\gamma_1 \prec \gamma_2\).

To cover the situation where something should not occur in between two event occurrences, the notation \((x, \neg y, z, \ldots) \iff \gamma_x \prec \gamma_z \land \neg \exists \gamma_y (\gamma_x \prec \gamma_y \prec \gamma_z)\) is used. For example, \((1, \neg 2, 3)\) means there is a \(\gamma_1 \prec \gamma_3\) and there are no \(\gamma_2\) in between \(\gamma_1\) and \(\gamma_3\).

Finally, to handle repeated event occurrences of the same type, the notation \((x_0, x_1, \ldots, x_n) \overset{\Delta}{=} \gamma_0 \prec \gamma_1 \prec \cdots \prec \gamma_n\) is used. For example, \((1^0, 2^1)\) means that there is \(\gamma_0^1 \prec \gamma_2^1\).

A '?' denotes that it is not significant if anything occurs. For example, \((2, \?)\) can mean \(\gamma_2, \gamma_2 \prec \gamma_1, \gamma_0^2 \prec \gamma_1^2\) etc. are the earliest available occurrences.

There are columns for representing overlapping contributing event occurrences. There is an “overlapping” column for each pair of event types contributing to a composite event (e.g., \(E_1; E_2\) has 1 column, whereas the expression \(N(E_1, E_2, E_3)\) has 3 columns). In an “overlapping” column 'T' denotes that the event occurrence overlap, 'F' denotes that they do not overlap, and '?' denotes that it does not matter or it is not applicable.

The “Rule” column specifies which rule is applicable in the case. No more than one rule should be applicable for each case, which can be verified by considering the range predicate.

Finally, the “U/I” column addresses in which set each event occurrence is added. Each column (1, 2, etc.) represents \(\gamma_1, \gamma_2\) etc. in the case. For example, in case 2, column 2 of the “U/I” column, the \(I_\omega\) implies that \(\gamma_2\) is added to this set for the composite event type \(E_1; E_2\).

The proof is performed by checking each row against the rules and then ensure that at most one rule is applicable at the same time. Each row represents a case where the hypothesis of a rule is true, and the rest of the rules are false. This table can be verified for all cases against the operator rules.

<table>
<thead>
<tr>
<th>#</th>
<th>Precedence</th>
<th>Overlap</th>
<th>Rule</th>
<th>U/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((1, 2, ?))</td>
<td>F</td>
<td>s</td>
<td>(U_\alpha)</td>
</tr>
<tr>
<td>2</td>
<td>((1, 2, ?))</td>
<td>T</td>
<td>s'</td>
<td>(I_\omega)</td>
</tr>
<tr>
<td>3</td>
<td>((2, ?))</td>
<td>?</td>
<td>s'</td>
<td>(I_\omega)</td>
</tr>
</tbody>
</table>

Table 6.1: Proof table for the sequence operator
The definition of sequence operator fulfills Theorem 6.2 on p. 119, since at least one of the candidate event occurrences in the rules \((s \text{ and } s')\) are assigned to used or invalidation sets.

Given that there are no infinite sequences of event occurrences of the same event type without any intermediate event occurrences of another event type, then there is also program liveness given the rule evaluation algorithm in Def. 6.11 on p. 109 and Lemma 6.4 on p. 132. In other words, eventually something good will happen (in our case, event composition will take place).

**Conjunction and Disjunction Operator**

The conjunction and disjunction event operators are presented here to illuminate why the event type sets are free variables in the event context predicates rather than using the \(\alpha(E)\) function (and correspondingly \(\omega(E)\) function) to derive them in an event context predicate. As mentioned earlier, the reason is that different event operator rules have different requirements on what event type should be considered. Consider Table 6.2 on the following page where the different event type sets of initiators and terminators are presented for usage rules that generate event occurrences. The rule definitions for conjunction are in Def. 6.45 on the following page and similarly for disjunction in Def. 6.46 on p. 139.

Obviously, it is possible to reconstruct the formal schema to allow writing context predicates using the functions \(\alpha(E)\) and \(\omega(E)\), however, this would require that the event operator is checked in the event context predicate (e.g., \((E \Rightarrow E_1; E_2 \Rightarrow \ldots) \land (E \Leftrightarrow E_1 \Leftrightarrow E_2 \Rightarrow \ldots) \land \ldots\)); the effect of this design would be that context predicates are more strongly coupled to the event specification language than is necessary. When a proof relating to the event context would be performed, this can result in more cases that need to be proved.
Formal Specification of Event Composition

Table 6.2: Event type sets for sequence, conjunction, and disjunction event operators

<table>
<thead>
<tr>
<th>Operator rule</th>
<th>$\mathcal{E}_\alpha$</th>
<th>$\mathcal{E}_\omega$</th>
<th>$\alpha(E)$</th>
<th>$\omega(E)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>${E_1}$</td>
<td>${E_2}$</td>
<td>${E_1}$</td>
<td>${E_2}$</td>
</tr>
<tr>
<td>$c[l]$</td>
<td>${E_1}$</td>
<td>${E_2}$</td>
<td>${E_1, E_2}$</td>
<td>${E_1, E_2}$</td>
</tr>
<tr>
<td>$c[r]$</td>
<td>${E_2}$</td>
<td>${E_1}$</td>
<td>${E_1, E_2}$</td>
<td>${E_1, E_2}$</td>
</tr>
<tr>
<td>$d[l]$</td>
<td>${E_1}$</td>
<td>${E_1, E_2}$</td>
<td>${E_1, E_2}$</td>
<td>${E_1, E_2}$</td>
</tr>
<tr>
<td>$d[r]$</td>
<td>${E_2}$</td>
<td>${E_1, E_2}$</td>
<td>${E_1, E_2}$</td>
<td>${E_1, E_2}$</td>
</tr>
</tbody>
</table>

Definition 6.45

Conjunction:

Let $E = E_1 \triangle E_2$ in the following rules and let

$\gamma \overset{\Delta}{\rightarrow} \Gamma(E, \text{start}((\text{span}(\gamma_1)), \text{end}((\text{span}(\gamma_2))))$ in the $c[l]$-rule:

$\forall E \forall \gamma_1, \gamma_2 \in \mathcal{G}$:

\[
\begin{align*}
\text{type}(\gamma_1) &= E_1 \land \text{type}(\gamma_2) = E_2 \\
X(E, \gamma_1, \gamma_2, \{E_1\}, \{E_2\}) \\
\gamma_1 \prec \gamma_2 \quad c[l]
\end{align*}
\]

$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_\alpha(E) \cup = \{\langle \gamma_1, \gamma \rangle\} \quad \mathcal{U}_\omega(E) \cup = \{\langle \gamma_2, \gamma \rangle\}$

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

$\forall E \forall \gamma_1, \gamma_2 \in \mathcal{G}$:

\[
\begin{align*}
\text{type}(\gamma_1) &= E_1 \land \text{type}(\gamma_2) = E_2 \\
X(E, \gamma_2, \gamma_1, \{E_2\}, \{E_1\}) \\
\gamma_2 \prec \gamma_1 \quad c[r]
\end{align*}
\]

$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_\alpha(E) \cup = \{\langle \gamma_2, \gamma \rangle\} \quad \mathcal{U}_\omega(E) \cup = \{\langle \gamma_1, \gamma \rangle\}$

$\square$

In other words, if the initiator occurrence is of the event type of the left operand, and the context predicate is true for both conjunctions that are initiated both with the left operand and the right operand, then a conjunction has occurred.

Proof of correctness is in Table 6.3 on the facing page. The definition of conjunction fulfills Theorem 6.2 on p. 119.
6.4 Discussion

Table 6.3: Proof table for the conjunction operator

<table>
<thead>
<tr>
<th>#</th>
<th>Precedence</th>
<th>Overlap</th>
<th>Rule</th>
<th>U/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 2, ?)</td>
<td>?</td>
<td>c[l]</td>
<td>Uα</td>
</tr>
<tr>
<td>2</td>
<td>(2, 1, ?)</td>
<td>?</td>
<td>c[r]</td>
<td>Uω</td>
</tr>
</tbody>
</table>

Definition 6.46
Disjunction:
Let $E = E_1 \lor E_2$ and let $\gamma \models \Gamma(E_1 \lor E_2, \text{[start(span($\gamma_1$), end(span($\gamma_1$)))]})$ in the $d[l]$-rule:

$$\forall E \forall \gamma_1 \in \mathcal{G} :$$

$$\begin{align*}
\text{type($\gamma_1$)} &= E_1 \\
X(E, \gamma_1, \gamma_1, \{E_1, E_2\}, \{E_1, E_2\}) &= \text{true} \\
\mathcal{G} \cup \{\gamma_1\} &\cup \mathcal{U}_a(E) \cup \{(\gamma_1, \gamma_1)\} &\cup \mathcal{U}_\omega(E) \cup \{(\gamma_1, \gamma_1)\}
\end{align*}$$

Let $\gamma \models \Gamma(E, \text{[start(span($\gamma_2$), end(span($\gamma_2$)))]})$ in the $d[r]$-rule:

$$\forall E \forall \gamma_1 \in \mathcal{G} :$$

$$\begin{align*}
\text{type($\gamma_1$)} &= E_2 \\
X(E, \gamma_1, \gamma_1, \{E_1, E_2\}, \{E_1, E_2\}) &= \text{true} \\
\mathcal{G} \cup \{\gamma_1\} &\cup \mathcal{U}_a(E) \cup \{(\gamma_1, \gamma_1)\} &\cup \mathcal{U}_\omega(E) \cup \{(\gamma_1, \gamma_1)\}
\end{align*}$$

Proof of correctness is found in Table 6.4 on the following page. The disjunction operator fulfills Theorem 6.2 on p. 119.

6.4 Discussion

By using this formalized schema for event composition, it is possible to define new context predicates and, thus, by symbolic manipulation reason about the effects of different contexts for different event expressions. For
example, it is possible to prove that two expressions are equivalent in specific contexts; that is, for the generation rules \( r_1 \) and \( r_2 \) the following holds: \( \forall s_i ( \text{condition}(s_i, r_1) \Leftrightarrow \text{condition}(s_i, r_2)) \) (cf. Theorem 6.1 on p. 118). This is used for the proof in Ch. 8.

The operator rules can be defined in terms of generating, using, and invalidating event occurrences. It is possible to add (e.g., temporal) constraints to define the semantics of early detection of timing constraint violations \[LMK98\]. The important part is that for each significant permutation of temporal orders as well as overlappings of earliest available occurrences, there is an applicable rule for that situation. All rules specifying the semantics of an event operator should fulfill Lemma 6.1 on p. 116.

Since the rules are defined using first-order predicate logic with special form of inference rules to allow consumption, it is possible to verify event composition in different mechanisms formally. For example, Algorithm 6.1 on p. 99 can be proven against the sequence rules using Hoare’s programming logic.

Only event operators that combine initiator and terminator event occurrences into composite event occurrences are considered in our schema. Our schema can be extended to handle event operators that combine more than initiator and terminator event occurrences, but these event operators (e.g., \( A^* \) in Snoop \[CM94, CKAK94\]) can result in composite event occurrences of arbitrary size and thus they cannot be used while ensuring resource-predictability of event composition \[Mel98a\].

In this work, a simple view of time has been chosen. It is possible to redefine the precedence operator to handle the timestamps suggested by, for example, Schwidersky et al. \[SHM95\]. This may entail new operator rules, if we want to handle relations such as “concurrent termination”. To be more specific, in Solicitor no two related event occurrences have equal timestamps and, therefore, this situation is not addressed in the operator rules of Solicitor. It is interesting to design a generic operator rule set for each operator, where Solicitor would be a specific instance. This issue is considered future work (in Section 15.3.3 on p. 313).

<table>
<thead>
<tr>
<th>#</th>
<th>Precedence</th>
<th>Overlap</th>
<th>Rule</th>
<th>U/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-2</td>
<td>?</td>
<td>( d[l] )</td>
<td>( U_\alpha, U_\omega )</td>
</tr>
<tr>
<td>2</td>
<td>1-2</td>
<td>?</td>
<td>( d[r] )</td>
<td>( U_\alpha, U_\omega )</td>
</tr>
</tbody>
</table>

Table 6.4: Proof table for the disjunction operator
A related issue to time is expiration. In our schema, the expiration predicate references \textit{now}; a fact that entails (i) that event composition must read the same time from the underlying operating system while event composition is taking place as well as that (ii) event composition must be invoked with respect to clock granularity so that both logical and temporal correctness is not violated. There are two problems that this solution manages: (i) the time of invocation of event composition, and (ii) event composition and its view of time. In case (i), if event composition is delayed too much, then, for example, the chronicle event context may result in incorrectly composed event occurrences due to that contributing event occurrences have expired due to delayed event composition. Such delays may lead to logical incorrectness in a system. In case (ii), if the clock increases its value while processing is taking place, then this may also lead to logical incorrectness in, for example, chronicle event context. N.B. it is strongly desirable that event composition can be completed within the clock granularity of a system, since this enables a system to be in a correct and consistent state at specific points in time.

6.4.1 Alternative Way to Expiration Semantics

An alternative way of adding the expiration in the context predicates is to add a new operator rule expressing the expiration as well as augmenting existing operator rules by using an expiration predicate. This new rule explicitly invalidates initiator occurrences when they expire in contrast to the implicit invalidation by the fact that the context predicate is not true for expired event occurrences (in other words, the occurrences cannot be selected).

However, it is considered more complicated to understand such a solution, since it involves more event operator rules. Moreover, a generic expiration rule for all event operators is hard to understand, since, for example, each event operator has a different set of contributing event types that can initiate it. In $E_1:E_2$ only $E_1$ event occurrences can initiate it, whereas in $E_1\triangle E_2$ both $E_1$ and $E_2$ can initiate it. Additionally, the proofs of correctness and completeness of event operator rules must be extended.

6.4.2 Algebraic Properties of Event Operators

Algebraic properties, of event operators such as reflexivity, symmetry, associativity, distributivity etc. depend on the event context. This issue has
not been addressed in this chapter, since processing has been in focus. As a consequence, legal transformations of event expressions are not addressed.

It is desirable to make event operators irreflexive. That is, given that $\mathcal{OP}_E$ is the set of all binary event operators $\forall E \forall \varnothing \in \mathcal{OP}_E \left( \neg (E \varnothing E) \right)$. The reason is that event operators constrain the order of contributing event occurrences and thus expressions such as $E_1; E_1$ are meaningless in event contexts such as recent. In the recent event context, such an expression would yield nothing since two event occurrences are never in sequence. To achieve meaningful expressions, binary event operators should be irreflexive. $N$-ary event operators also define semantics in terms of order between the contributing event types and, therefore, these event operators should be applied to distinct event types. Sometimes it is desirable to express some property of recurring event occurrences such as minimum interarrival time. One solution is to treat the same base event type with a relative order to each other as different event types. For example, $E = E_1; \langle \text{left.order} = \text{right.order} + 1 \rangle E_1$ would $E_1$ be treated as two different event types within the scope of $E$. This is considered future work (in Section 15.3.13 on p. 317).

6.5 Related Work on Formalization of Event Composition

Since our schema separates the context semantics from the event operator semantics, it is desirable to evaluate this against other existing approaches. This entails that the approaches are evaluated in terms of: (i) supported event contexts, (ii) explicit/implicit support for the context, and (iii) the scope of reasoning concerning a property with respect to event contexts. The supported contexts indicate the expressive power of the formalism. If the support for contexts is implicit, then it is not necessary to explicitly realize the context semantics in the chosen mechanism. Finally, the context scope tells us if it is possible to prove a property across event contexts or not.

First, the formalization of Snoop based on event histories [CYY98] and as an extension of the work by Galton et al. [GA01] by Adaikkalavan [Ada02] is considered. Moreover, ODE [JMS92, GJM93, GJS93] and SMILE [Jae97] are considered. Then, monitoring of real-time logic [ML97a, ML97b, LMK98, LM99] is addressed. This is followed by a comparison to active database formalisms as well as related work in deductive databases. Finally, attempts to specify event contexts in a different way is briefly addressed.
6.5 Related Work on Formalization of Event Composition

6.5.1 Formalization of Snoop over Event Histories

As mentioned, Chakravarthy et al. [CM94, CKAK94, CYY98] only consider the termination time not the interval of an event. Their notation is that $E(t)$ means that an event $E$ has terminated at time $t$ [CM94, CKAK94], that is, $\forall t' (E(t') \iff \exists t (t \leq t' \land \text{occ}(E!, [t, t']))).

Similarly to this work, they use event histories, where the symbol $H$ represents the event history (where $H \subseteq G$) such that $E[H] = \{\gamma | \gamma \in G \land \text{type}(\gamma) = E\}$. In other words, $E[H]$ is a set of event occurrences of event type $E$. Their formalism [CYY98] is based on event histories as is the case in our approach. They support the same contexts as in our approach: recent, chronic, continuous, and general contexts. Similarly to our approach, this support is implicit, however, their approach does not consider processing, only the outcome of an event history.

In their formalization, Chakravarthy et al. [CYY98] define event operators in terms of event histories in the general event context. For example, let $t_1 = \lambda \min(\text{start}(\text{span}(\gamma_1)), \text{start}(\text{span}(\gamma_2)))$ (i.e., the minimum start time of two event occurrences) and $t_2 = \lambda \max(\text{end}(\text{span}(\gamma_1)), \text{end}(\text{span}(\gamma_2)))$ (i.e., the maximum end time of two event occurrences) in

$$(E_1 \triangle E_2)[H] = \{\gamma | \gamma_1 \in E_1[H] \land \gamma_2 \in E_2[H] \land \gamma = \Gamma(E, [t_1, t_2])\}$$

Then, they define a filter $\sigma$ for each event context that applied to $E[H]$ (written $\sigma \cdot E[H]$) gives the result of event type in the event context $\sigma$. In the formalism used in our work, $\sigma \cdot E[H] = \{\gamma | \gamma \in E[H] \land \sigma\}$. The filter for recent event context rewritten using our notation:

\begin{align*}
\forall \gamma' \in E[H] (\gamma' \neq \gamma' \land \text{end}(\text{span}(\gamma')) = \text{end}(\text{span}(\gamma)) \Rightarrow \\
\text{end}(\text{span}(\alpha(\gamma))) \geq \text{end}(\text{span}(\alpha(\gamma')))) & \} \text{ Case 1} \\
\wedge
\end{align*}

\begin{align*}
\forall \gamma' \in E[H] (\gamma \neq \gamma' \land \text{end}(\text{span}(\alpha(\gamma))) = \text{end}(\text{span}(\alpha(\gamma')))) & \} \text{ Case 2} \\
\wedge
\end{align*}

\begin{align*}
\forall \gamma' \in E[H] (\gamma \neq \gamma' \land \text{end}(\text{span}(\gamma)) > \text{end}(\text{span}(\gamma')) \Rightarrow \\
\text{end}(\text{span}(\alpha(\gamma))) > \text{end}(\text{span}(\alpha(\gamma')))) & \} \text{ Case 3}
\end{align*}

3The general context is frequently called unrestricted context.

4Since they lack functions such as 'end', 'span' etc., their expressions contain more bound variables making them harder to read.
The three cases in the filter are depicted in Fig. 6.4. To explain the filter, consider the following explanation of each case:

1. For all event occurrences with the same termination timestamp, the one that holds true in the filter is the one with the most recent initiator event occurrence before it.

2. For all event occurrences with the same initiation timestamp, the filter is true. N.B., the original definition by Chakravarthy et al. [CYY98] states: \( \forall \gamma' \in E[H] (\gamma \neq \gamma' \land \text{end(span}(\alpha(\gamma))) = \text{end(span}(\alpha(\gamma'))) \Rightarrow \text{end(span}(\omega(\gamma))) \leq \text{end(span}(\omega(\gamma')))) \). In other words, the filter is only true for the first event occurrence with the same initiating event occurrence. However, this is incorrect (except for non-overlapping recent context), since initiator occurrences may be reused in successive composite event occurrences.

3. For all event occurrences \( \gamma \) that terminate after another event occurrence \( \gamma' \), then \( \gamma \) should be initiated after \( \gamma' \) has terminated.

N.B. the “conjunction” of cases implies, obviously, that if any case is false, the currently considered event occurrence \( \gamma \) is a result of \( \sigma \cdot E[H] \).

![Figure 6.4: Depiction of recent filter in Snoop](image-url)

Compare this filter definition to the simplicity of the recent event context predicate Def. 6.40 on p. 129. Since separation of concern is an important part of the formalism presented in this work, the separate parts are more manageable. For example, the specification of non-overlapping event contexts require an addition to each event context, whereas in the formalism
6.5 Related Work on Formalization of Event Composition

proposed by Chakravarthy et al. the filters may have to be redefined. However, the cost is that standard proof techniques may not be used directly, but have to be adapted for the formalism. Despite this fact, the formalism in this work has been used to prove an equivalence within the Solicitor language (in Ch. 8). In contrast, the complexity of the general event context predicate Def. 6.42 on p. 130 is greater compared to the simplicity of the filters by Chakravarthy et al. However, the general event context is the most intuitive to understand and least useful for event composition; thus, we consider this additional complexity for general event context to be a fair price for the simplicity of the other event context predicates.

The idea of applying filters is compelling, since it goes from the general event context to a specific event context applied to a subset of $G$. Unfortunately, these filters are complicated. In contrast, the formalized schema presented here separates used and invalidated contributing event occurrences and, thereby, it is possible to use the separation for proofs. For example, we can prove that the context predicates selects the correct event occurrences if and only if the operator rules are properly designed.

6.5.2 Formalization of Snoop using First Order Predicate Logic

As mentioned, Adaikkalavan [Ada02] has worked on an extension to the work by Galton et al. [GA01] where event contexts are included. First of all, it does not address processing. Moreover, there is no clear separation of operator part of the semantics and the context part. For example, the sequence operator in recent event context is defined as follows:

$$
\forall[t, t'] (occ(E_1; E_2, [t, t']) \Leftrightarrow \forall[t_1, t'_1], [t_2, t'_2] (t = t_1 \land t' = t'_2 \Rightarrow occ(E_1, [t_1, t'_1]) \land occ(E_2, [t_2, t'_2]) \land (t'_1 < t_2) \land \neg \exists[t_3, t'_3] (occ(E_1, [t_3, t'_3]) \land t'_1 < t'_3 \leq t')))
$$

This is an interpretation of the definition by Adaikkalavan [Ada02], since it is not complete. For example, he uses $occ(E, [t, t'])$ as well as $occ(\gamma, [t, t'])$ where $\gamma = \Gamma(E, [t, t'])$ without defining the precise meaning of the different predicates. Moreover, event histories from [CYY98] are mixed into the expression without being necessary. For example, instead of asserting the facts $occ(E_1, [1, 1]), occ(E_1, [2, 2])$ they state that the event history $E_1[H] = \{\Gamma(E_1, [1, 1]), \Gamma(E_2, [2, 2])\}$ and use this in the expression as $\forall \gamma \in E_1[H] \ldots \ldots$ instead of $\forall[t_1, t'_1] (occ(E_1, [t_1, t'_1]) \Rightarrow \ldots)$. 
There are some interesting ideas, but they need to be further refined. However, since first order predicate logic cannot handle consumption of truths, it is difficult to handle the chronicle event context in such a formalism.

### 6.5.3 Interval-Based Algebra for Restricted Event Detection

As mentioned, Carlson et al. [CL03] has introduced an interval-based algebra for restricted event detection. Their work is rigorous in terms of model-theoretic reasoning about their event algebra. In the ontology used in their work, they have introduced a novel event context that preserves intuitive algebraic properties.

In a similar vein to the first approach by Chakravarthy et al. [CYY98], they define event histories (that they call event streams). Then, they define their operators in terms of these event streams as well as a fixed event context. In contrast to the event context predicates in this work, their event context is dependent on the event operator. The reason for this dependence is to preserve algebraic properties of the event specification language. The definition of their context is presented in the format of this work in Def. 6.47. As can be seen, the sequence operator has a separate treatment from all other event operators.

**Definition 6.47**

\[\begin{align*}
\text{carlson}_\alpha(E, \gamma_\alpha, E_\alpha) & \triangleq \gamma_\alpha \in V_{ALID\alpha} \\
\text{carlson}_\omega(E, \gamma_\omega, E_\omega) & \triangleq \gamma_\omega \in V_{ALID\omega} \\
\text{shortest\_interval}_\omega(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega) & \triangleq \\
& \forall \gamma (\gamma \in V_{ALID\alpha} \land \hat{\alpha}(\gamma_\alpha) < \hat{\alpha}(\gamma) \land \gamma_\omega \neq \gamma \Rightarrow \gamma_\omega < \gamma) \land \\
& \forall \gamma (\gamma \in V_{ALID\omega} \land \gamma_\omega > \gamma \land \hat{\alpha}(\gamma_\alpha) \neq \hat{\alpha}(\gamma) \Rightarrow \hat{\alpha}(\gamma_\alpha) > \hat{\alpha}(\gamma)) \\
\text{formed\_composite\_event}(E, \gamma, \gamma') & \triangleq \\
& \exists \gamma'' (\text{type}(\gamma'') = E \land \text{initiate}(\gamma, \gamma'') \land \text{terminate}(\gamma', \gamma''))
\end{align*}\]
6.5 Related Work on Formalization of Event Composition

\[ carlson_{\alpha\omega}(E, \gamma_{\alpha}, \gamma_{\omega}, E_{\alpha}, E_{\omega}) \overset{\Delta}{=} \]
\[ \forall E', E'' \ (E = E'; E'' \land \text{type}(\gamma_{\alpha}) = E'' \land \text{type}(\gamma_{\omega}) = E'') \Rightarrow \]
\[ \text{shortest}_{\text{interval}}(E, \gamma_{\alpha}, \gamma_{\omega}, E_{\alpha}, E_{\omega}) \land \]
\[ \forall \gamma, \gamma' \ (\gamma \in V_{\text{ACTD}_{\alpha}} \land \gamma' \in V_{\text{ACTD}_{\omega}} \land \]
\[ \text{shortest}_{\text{interval}}(E, \gamma, \gamma', E_{\alpha}, E_{\omega}) \land \gamma \prec \gamma_{\alpha} \land \gamma' \prec \gamma_{\omega} \Rightarrow \]
\[ \text{formed}_{\text{composite}}_{\text{event}}(E, \gamma, \gamma') \]
\[ \forall E', E'' \ (E \neq E'; E'' \Rightarrow \]
\[ \neg \exists \gamma \in V_{\text{ACTD}_{\alpha}} \ (\gamma_{\alpha} \neq \gamma \land \gamma \prec \gamma_{\omega} \land \hat{\alpha}(\gamma_{\alpha}) \prec \hat{\alpha}(\gamma)) \]
\[ \square \]

In other words, if the principal event operator of an event type is sequence, then the context predicate is true for the earliest non-used pair of initiator and terminator occurrence that has not been used already. When their are overlapping possibilities, the pair with the shortest span should be used. If the principal event operator is not a sequence, then the event context is similar to the recent event context; the difference between the recent context and Carlson et al.’s context is that they compare initiators of contending initiator occurrences rather than the terminator of these contending initiators.

6.5.4 ODE and SMILE

ODE [JMS92, GJM93, GJS93] and SMILE [Jae97, JO98] have a history-based formalism similar to Snoop. However, none of these approaches address processing in such a way that it can be used to discuss processing for real-time purposes.

6.5.5 Monitoring of Real-Time Logic

As addressed in Ch. 4, a unified approach between event composition and monitoring using real-time logic has been proposed by Liu et al. [LMK98]. They have proven properties such as resource-predictability, but only in the supported event contexts: recent and chronicle. Their support for contexts is implicit. Moreover, their specification is at a lower-level since their specifications require more detail than specifications in Solicitor.
6.5.6 Active Database Formalisms

In contrast to this work, Paton et al. [PD99], Campin et al. [CPW97] and Bertossi et al. [BP99] have developed separate formalisms that address the entire rule processing of an active database. In a survey [PCFW95], different approaches to formalization have been considered. Unfortunately, none of these formalisms address event composition in such a way that it can be used for reasoning about the processing.

6.5.7 Event Composition Formalism Based on Deductive Databases

Motakis et al. [MZ97] presents the event pattern language (EPL) based on DataLog1S, a database-oriented variant of Prolog. The advantage is that their rules can be executed in a DataLog1S engine. This can be performed in an efficient manner without backtracking. In EPL, they have compared and contrasted Snoop, ODE, and SAMOS and demonstrated that it is possible to define all their semantics within the scope of EPL. However, the event contexts are not parameterized. Moreover, since they only consider what is happening within a transaction, the relation between the initiator and terminator occurrence is not explicit as in the formalism presented in this work. For example, the result of event composition in recent event context in their work is only the last composed event occurrence. Since processing may involve the preceding composite event occurrences in recent context, then it is desirable to have a formalism that addresses this fact when processing should be defined.

It is desirable to do the same kind of comparison using our formalized schema (in Section 15.3.1 on p. 312), since it may reveal other properties not visible in the work by Motakis et al. (considered future work in Section 15.3.1 on p. 312).

6.5.8 Decorating Event Specification Languages

Zimmer et al. [ZU99] present a structured and systematic evaluation (which they call a formalism) that, instead of using event contexts associated with event operators, use decorators associated with the contributing event types (a similar attempt has been made by Zhang et al. [ZU96]). For example, they specify the sequence $E_1:(context: recent)E_2$ as \textit{last: shared: }$E_1:exclusive: E_2$. Where \textit{last} implies select the last occurrence
of the contributing event type, *shared* implies that an event occurrence may be reused, and *exclusive* implies that an event occurrence may only be used once. However, some combinations are not meaningful, for example, \(\text{first}:\text{shared}:E_1;\text{first}:\text{shared}:E_2\) would never ever do anything but repeatedly generate the same composite event occurrence. The structured evaluation of Zimmer et al. is not as precise as our formalism. For example, the relation between initiator and terminator for different event contexts is not as clear as in this work. Zimmer et al. has made an extensive comparison that would be useful to validate using our formalized schema in this work (considered future work in Section 15.3.1 on p. 312).

### 6.6 Summary

To summarize, in this chapter a formalized schema for defining context predicates and (event) operator rules for an event specification language has been defined. The important issue is that there is a clear separation between scheduling of operator rules applied to event types, the selection of event occurrences of contributing event types to test, and the test if the selected event occurrences fulfill the constraint of the event operator. This clear separation allows separate reasoning for the different parts as well as combined reasoning.

Necessary and sufficient constraints on context predicates and operator rules have been defined. The power of the schema has been demonstrated by defining the Solicitor event specification language.
Chapter 7

Realization Issues

“In theory’ is an English phrase that means ‘not really’.”

/David L. Parnas

In this chapter, we present realization issues that are necessary to give the investigation of resource-predictability as well as the time complexity study in the following chapters substance in terms of how the theory relates to implementation. The problem addressed is to demonstrate that a formalism of chapter Ch. 6 can be used in practice. To demonstrate this, necessary invariants for proving (or generating) an implementation are derived from the formal specification of Solicitor.

For simplicity and without loss of generality, we only consider contributing event types that can initiate or terminate an event type. The main reason for this simplification is that all event operators have contributing event types that are initiators or terminators, while there may be optional event types: (i) whose event occurrences either inhibit or allow composition of a composite event occurrence (e.g., aperiodic or non-occurrence), or (ii) whose event occurrences are aggregated as interiors together with the initiator and terminator event occurrence into a composite event occurrence. Albeit the fact that monitoring of real-time logic (e.g., [LMK98]) and Chronicle recognition (e.g., [Dou96]) differs from other event specification languages since the whole real-time logic expression and, correspondingly, the whole Chronicle is
evaluated rather than evaluating each operator in the Chronicle separately
(in Section 4.9.3 on p. 78), the resulting event occurrences are still the same.

As addressed in Section 6.2 on p. 102, we also assume that event
occurrences are stable and totally ordered before they are sent to the task
performing event composition. That is, the task performing event composit-
donot have to contain mechanisms of unstable event occurrences or
non-ordered event occurrences.

7.1 Connector Set Abstraction for Event
Composition

As mentioned earlier (in Section 4.9 on p. 75), the filtered event log is
assumed to be realized as connector sets between event types and their con-
tributing event types, where each connector set contains event occurrences
to enable event composition of the contributed event type. Restricted sets
(e.g., queues) are used to realize the filtered event log in the approaches
addressed in this work. The advantage of set representations are that they
allow extensions for, for example, distribution; a desirable extension in dis-
tributed systems can be to weaken the order requirement, a situation that
implies that a queue abstraction is incorrect since event occurrences may be
delivered to the event composer out of order. Moreover, properties demon-
strated on a set representation can hold on restricted set representations
such as queues.

Let $Q_{E'}(E)$ be the connector set between the contributing event type $E'$
and the event type $E$. The general definition of a connector set is (defined
by its invariant):

**Definition 7.1**

General connector set invariant:

\[
\forall E, E' \left( E' \in \text{all_operands}(E) \Rightarrow Q_{E'}(E) \subseteq G \land \forall \gamma \in Q_{E'}(E) \left( \text{type}(\gamma) = E' \right) \right) \land \\
\forall E, E' \left( E' \notin \text{all_operands}(E) \Rightarrow Q_{E'}(E) = \emptyset \right)
\]

\[\square\]

In other words, connector sets are only defined between an event type and the
contributing event types of the event type’s principal event operator. More-
7.1 Connector Set Abstraction for Event Composition

over, connector sets are subsets of all generated event occurrences, where the event type of all event occurrences in a connector set is of the contributing event type in question. For example, \( E = E_1; E_2 \) has the connector sets \( Q_{E_1}(E) \) and \( Q_{E_2}(E) \). N.B. the general connector set definition only states what event type the event occurrences in the connector set must fulfill. The specific definition depends on the event contexts.

A potential problem with this notation is that expressions such as \( E_1; E_1 \) are ambiguous. However, as discussed earlier (in Section 6.4.2 on p. 141), such event expressions are disallowed.

To be more specific, the problem addressed in this section is to formulate a general invariant, parameterized on event contexts, that is true when the connector sets contain the correct set of event occurrences. Before giving the full definition of the connector set invariant, the following examples illuminate some of the problems of formulating such a general parameterized invariant.

**Example 7.1**

Given \( E_1;(\text{context: recent})E_2 \) and the event history in Fig. 4.2 on p. 58, in the case where event composition is implicitly invoked after each primitive event is signalled, the invariant is informally that (i) \( Q_{E_1}(E) \) contains only the most recent initiator occurrence that are of event type \( E_1 \) and (ii) \( Q_{E_2}(E) \) is empty since, in implicit invocation, no terminator occurrence is ever stored. The reason for the part (i) of the invariant is that only the most recent initiator occurrence needs to be stored in the connector set to form composite event occurrences in recent event context. To be more specific: during time 1 to time 3, \( Q_{E_1}(E) = \{\Gamma(E_1, [1,1])\} \); at time 4 \( Q_{E_1}(E) = \{\Gamma(E_1, [4,4])\} \); and at time 5 and onwards \( Q_{E_1}(E) = \{\Gamma(E_1, [5,5])\} \).

In the case where evaluation is explicitly invoked after an arbitrary number of primitive event occurrences has been signalled, then the invariant is informally that \( Q_{E_1}(E) \) contains the initiator occurrence (of event type \( E_1 \)) that was most recent at the last evaluation as well as initiator event occurrences that have occurred after the last evaluation. Additionally, \( Q_{E_2}(E) \) contains the terminator occurrences that have occurred after the last evaluation. Assuming that evaluation takes place at time 5, then the connector sets \( Q_{E_1}(E) = \{\Gamma(E_1, [1,1]), \Gamma(E_1, [4,4]), \Gamma(E_1, [5,5])\} \) and \( Q_{E_2}(E) = \{\Gamma(E_2, [2,2]), \Gamma(E_2, [3,3])\} \) before the evaluation, and \( Q_{E_1}(E) = \{\Gamma(E_1, [5,5])\} \) and \( Q_{E_2}(E) = \emptyset \) after the evaluation.
Example 7.2
Given $E_1;\langle\text{context: chronicle}\rangle E_2$ and the event history in Fig. 4.3 on p. 58, in the case where event composition is implicitly invoked after each primitive event is signalled, the invariant is informally that (i) $Q_{E_1}(E)$ contains only the event occurrences of event type $E_1$ that has neither been used nor invalidated and (ii) $Q_{E_2}(E)$ is empty since terminators are immediately processed. The reason for the part (i) of the invariant is that only unconsumed occurrences need to be stored in the connector set to form composite event occurrences in chronicle event context. To be more specific: at time 1, $Q_{E_1}(E) = \{\Gamma(E_1,[1,1])\}$; during time 2-3 $Q_{E_1}(E) = \emptyset$; at time 4 $Q_{E_1}(E) = \{\Gamma(E_1,[4,4])\}$; at time 5 $Q_{E_1}(E) = \{\Gamma(E_1,[4,4]), \Gamma(E_1,[5,5])\}$; and at time 6 $Q_{E_1}(E) = \{\Gamma(E_1,[5,5])\}$.

The case where evaluation is explicitly issued after an arbitrary number of primitive event occurrences has been signalled is treated the same way as in the previous example.

7.1.1 Invariant for Implicit Invocation

In order to formulate a general parameterized invariant, addressed issues in the examples (e.g., implicit/explicit invocation, parameterization over event contexts) must be taken into account. Since the invariant for explicit invocation is an extension to the invariant for implicit invocation, the invariant of implicit invocation is presented first. The general invariant uses Def:s 6.30 to 6.31 on pp. 125–126 and Def:s 6.32 to 6.33 on p. 126 in Section 6.3.3 on p. 124.

Definition 7.2
Connector set invariant for implicit invocation: Let $E$ be an event expression, where $E' \in \alpha(E)$ (in other words, $E'$ is a potential initiator type) and, similarly, $E'' \in \omega(E)$; let $X = \text{context}(E)$, let $\forall_{\forall \forall_{\forall T D'}} \overset{\Delta}{=} \text{all} of\text{ types}\((E')\) \setminus \mathcal{I}_{\forall \forall_{\forall T C a}}$ by using \text{all} of\text{ types} (in Def. 6.35 on p. 128), let $\forall_{\forall \forall_{\forall T D''}} \overset{\Delta}{=} \text{all} of\text{ types}\((E'')\) \setminus \mathcal{I}_{\forall \forall_{\forall T C a}}$, and assume that the dummy terminator occurrence $\Gamma(E'',[\infty,\infty]) \in \mathcal{G}$ to avoid an empty set of terminators.
Moreover, 

\[
\text{cur}(E, E', E'') = \{ \gamma^a | \gamma^a \in V_{\text{ALID'}} \land \exists \gamma^b \in V_{\text{ALID''}} (X(E, \gamma^a, \gamma^b, \{E''\}, \{E''\})) \}
\]

\[
is_{\text{first-available}}(\gamma) \triangleq \forall \gamma^d \in \text{cur}(E, E', E'') (\gamma \neq \gamma^d \Rightarrow \gamma \prec \gamma^d)
\]

\[
pot_{\text{init}}(\gamma) \triangleq \gamma \in V_{\text{ALID'}} \land \\
\exists \gamma^c \in \text{cur}(E, E', E'') (is_{\text{first-available}}(\gamma^c) \land (\gamma = \gamma^c \lor \gamma > \gamma^c))
\]

then:

\[
\forall E, E', E'' (E' \in \alpha(E) \land E'' \in \omega(E) \land E' \neq E'') \Rightarrow \\
Q(E) = \{ \gamma | pot_{\text{init}}(\gamma) \} \land \\
\forall E, E' (E' \in \omega(E) \land E' \notin \alpha(E)) \Rightarrow \\
Q(E) = \{ \gamma | \gamma \in G \land \text{type}(\gamma) = E \} \setminus (U_{\omega}(E) \cup I_{\omega}(E)) = \emptyset
\]

\[
\square
\]

In other words, the connector set between \( E' \) and \( E \) are all the potential initiator occurrences that are valid according to the context predicate \( pot_{\text{init}} \). This predicate is true for all occurrences of event type \( E' \) that either are equal to or succeed the occurrences that are in \( \text{cur}(E, E', E'') \) according to the \( pot_{\text{init}} \) predicate. The set \( \text{cur}(E, E', E'') \) contains initiator occurrences that fulfill the context predicate \( X \). The event occurrence \( \Gamma(E'[, \infty, \infty]) \) is necessary, since the context predicate requires both an initiator occurrence and a terminator occurrence to be evaluable. For contributing event types that can only be terminators, the connector set is empty. For contributing event types that can neither initiate nor terminate, this invariant is undefined. N.B. for contributing event types that can neither be initiators or terminators, the general invariant Def. 7.1 on p. 152 holds.

**Proof of Invariant for Implicit Invocation for All Event Contexts**

In this proof, there is a case for each event context followed by an induction proof for all possible event contexts whose event operators only combine initiators and terminators. Each proof starts with the substitution of the context predicate in the definition of the \( \text{cur}(E, E', E'') \). Since each context predicate consists of three parts, the \( \text{cur}(E, E', E'') \) in Def. 7.2 on the facing page can be rewritten as in Eq. 7.1 on the following page by using Def. 6.34 on p. 127.
\[
cur(E, E', E'') = \\
\{\gamma^a | \gamma^a \in V_{ALID'} \land \\
\exists \gamma^b \in V_{ALID''} \\
(X_\alpha(E, \gamma^a, \{E'\}) \land \\
X_\omega(E, \gamma^b, \{E''\}) \land \\
X_{\alpha\omega}(E, \gamma^a, \gamma^b, \{E', \{E''\}))}\}
\]

(7.1)

Lemma 7.1
In all event contexts, \( V_{ALID'} = V_{ALID_\alpha} \) and \( V_{ALID''} = V_{ALID_\omega} \) for the connector set invariant. \( V_{ALID_\alpha} \) and \( V_{ALID_\omega} \) are defined in Def. 6.38 on p. 128.

Proof: Consider how the event context predicate is used in the definition of \( cur(E, E', E'') \) in Def. 7.2 on p. 154: \( \mathcal{E}_\alpha = \{E'\} \). Therefore, the following holds: \( V_{ALID_\alpha} = All_\alpha \setminus Inv_{AL_\alpha} = all\_of\_types(\mathcal{E}_\alpha) \setminus Inv_{AL_\alpha} = all\_of\_types(\{E'\}) \setminus Inv_{AL_\alpha} \). Thus, \( V_{ALID'} = V_{ALID_\alpha} \). The proof for \( V_{ALID''} = V_{ALID_\omega} \) is left for the reader. N.B. in an operator rule this lemma may not hold, since \( \mathcal{E}_\alpha \) may contain more than one event type; it is only in the scope of the connector set invariant that this lemma holds. Q.E.D.

Definition 7.3
Available set definitions for connector set invariant:
\[
\begin{align*}
UNCONS' &= V_{ALID'} \setminus (CONS_\alpha \cup CONS_\omega) \\
UNCONS'' &= V_{ALID''} \setminus (CONS_\alpha \cup CONS_\omega)
\end{align*}
\]

\( \square \)

Lemma 7.2
Given Def. 7.3, then \( UNCONS' = UNCONS_\alpha \) and \( UNCONS'' = UNCONS_\omega \) for event contexts used in Def. 7.2 on p. 154.

Proof: Given Lemma 7.1, \( UNCONS' = V_{ALID_\alpha} \setminus (CONS_\alpha \cup CONS_\omega) \) holds. This is the same definition as \( UNCONS_\alpha \) (in Def. 6.38 on p. 128). The proof that \( UNCONS'' = V_{ALID_\omega} \setminus (CONS_\alpha \cup CONS_\omega) \) is left for the reader. Q.E.D.
7.1 Connector Set Abstraction for Event Composition

Lemma 7.3

Distributivity of \('\land' over '∃' (cf. [GS93, Theorem 9.23]):

\[ \exists \gamma \in X (P \land Q(\gamma)) \iff P \land \exists \gamma \in X (Q(\gamma)) \]

is true iff \(\gamma\) does not occur in \(P\).

Proof: Given that \(X = \{\gamma_0, \gamma_1, \ldots, \gamma_n\}\), then the following holds:

\[ \exists \gamma \in X (P \land Q(\gamma)) \iff (P \land Q(\gamma_0)) \lor (P \land Q(\gamma_1)) \lor \ldots \lor (P \land Q(\gamma_n)) \iff P \land \exists \gamma \in X (Q(\gamma)) \]

if \(\gamma\) does not occur in \(P\). Q.E.D.

Lemma 7.4

Connector set invariant for implicit invocation holds for recent event context:

In the recent event context, \(Q_{E'}(E)\) contains only the most recent occurrence of a \(E'\) if \(E' \in \alpha(E)\).

Proof: By using Def. 6.34 on p. 127 and letting \(X = \text{recent}\), \(\gamma_a = \gamma^a\), \(\mathcal{E}_a = \{E'\}\), \(\mathcal{E}_\omega = \{E''\}\), and \(\gamma_\omega = \gamma^b\) then Eq. 7.1 on the facing page can be rewritten as Eq. 7.2.

\[
\text{cur}(E, E', E'') = \\
\{\gamma^a | \gamma^a \in V_{ALID'} \land \\
\exists \gamma^b \in V_{ALID''} \\
(\text{recent}_a(E, \gamma^a, \{E'\}) \land \\
\text{recent}_a(E, \gamma^b, \{E''\}) \land \\
\text{recent}_\omega(E, \gamma^a, \gamma^b, \{E', \{E''\}\}))\} \quad (7.2)
\]

By replacing the different parts of \(\text{recent}\) (in Def. 6.40 on p. 129), we obtain Eq. 7.3 on the following page.
\[ \text{cur}(E, E', E'') = \]
\[ \{ \gamma^a | \gamma^a \in \text{VALID}' \land \exists \gamma^b \in \text{VALID}'' \]
\[ (\gamma^a \in \text{VALID}_\alpha \land \forall \gamma \in \text{UNCONS}_{\omega} (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma) \land \gamma^b \in \text{UNCONS}_{\omega} \land \neg \exists \gamma \in \text{VALID}_\alpha (\gamma^a \neq \gamma \land \gamma^a \prec \gamma \prec \gamma^b) \}\]  \hspace{1cm} (7.3)

Since Lemma 7.1 on p. 156 and Lemma 7.2 on p. 156 holds, then \( \text{VALID}_\alpha \) and \( \text{VALID}_\omega \) can be substituted with \( \text{VALID}' \) and \( \text{VALID}'' \). Likewise, the sets \( \text{UNCONS}_\alpha \) and \( \text{UNCONS}_\omega \) can be replaced with \( \text{UNCONS}' \) and \( \text{UNCONS}'' \). These substitutions give Eq. 7.4

\[ \text{cur}(E, E', E'') = \]
\[ \{ \gamma^a | \gamma^a \in \text{VALID}' \land \exists \gamma^b \in \text{VALID}'' \]
\[ (\gamma^a \in \text{VALID}' \land \forall \gamma \in \text{UNCONS}'' (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma) \land \gamma^b \in \text{UNCONS}'' \land \neg \exists \gamma \in \text{VALID}' (\gamma^a \neq \gamma \land \gamma^a \prec \gamma \prec \gamma^b) \}\]  \hspace{1cm} (7.4)

Eq. 7.4 can be reduced to Eq. 7.5 due to Lemma 7.3 on the previous page.

\[ \text{cur}(E, E', E'') = \]
\[ \{ \gamma^a | \gamma^a \in \text{VALID}' \land \exists \gamma^b \in \text{VALID}'' \]
\[ (\forall \gamma \in \text{UNCONS}'' (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma) \land \gamma^b \in \text{UNCONS}'' \land \neg \exists \gamma \in \text{VALID}' (\gamma^a \neq \gamma \land \gamma^a \prec \gamma \prec \gamma^b) \}\]  \hspace{1cm} (7.5)

Since \( \text{UNCONS}'' \subseteq \text{VALID}'' \), then Eq. 7.6 on the facing page is obtained.
7.1 Connector Set Abstraction for Event Composition

\[
\text{cur}(E, E', E'') = \\
\{ \gamma^a | \gamma^a \in \mathcal{ALID}' \land \\
\exists \gamma^b \in \mathcal{UNCONS}'' \\
(\forall \gamma \in \mathcal{UNCONS}'' (\gamma^b \neq \gamma \Rightarrow \gamma^b < \gamma)) \land \\
\neg \exists \gamma \in \mathcal{ALID}' (\gamma^a \neq \gamma \land \gamma^a < \gamma < \gamma^b) \} 
\] (7.6)

By letting \(\gamma'_\omega \in \mathcal{UNCONS}''\) (where \(\gamma'_\omega = \Gamma(E'', [\infty, \infty])\) be the earliest available non-consumed terminator such that for \(\gamma^b = \gamma'_\omega\) the subpredicate \(\forall \gamma \in \mathcal{UNCONS}'' (\gamma^b \neq \gamma \Rightarrow \gamma^b < \gamma)\) is true. In this case, \(\text{cur}(E, E', E'')\) can then be rewritten in the following way: \(\text{cur}(E, E', E'') = \{ \gamma^a | \gamma^a \in \mathcal{ALID}' \land \\
\neg \exists \gamma \in \mathcal{ALID}' (\gamma^a \neq \gamma \land \gamma^a < \gamma < \gamma^b) \} \). There are two cases for the earliest available initiator occurrence \(\gamma'_a \in \mathcal{ALID}'\): let \(\gamma^a = \gamma'_a\) (i) fulfill \(\forall \gamma \in \mathcal{ALID}' (\gamma^a \neq \gamma \land \gamma^a < \gamma < \gamma'_a)\) or (ii) not fulfill \(\forall \gamma \in \mathcal{ALID}' (\gamma^a \neq \gamma \land \gamma^a < \gamma < \gamma'_a)\). In case (i), \(\text{cur}(E, E', E'') = \{ \gamma'_a \} \) since it is the most recent event occurrence. In case (ii), \(\text{cur}(E, E', E'') = \emptyset\) since there are no initiator occurrences before \(\gamma'_a\).

The predicate \(\text{pot\_init}\) can be rewritten as: \(\text{pot\_init}(\gamma) \triangleq \gamma \in \mathcal{ALID}' \land \\
\exists \gamma^c \in \text{cur}(E, E', E'') (\text{is\_first\_available}(\gamma^c) \land (\gamma = \gamma^c \lor \gamma > \gamma^c))\). In case (i), this definition can be reduced to: \(\text{pot\_init}(\gamma) \triangleq \gamma = \gamma'_a \lor \gamma > \gamma'_a\), since the predicate \(\text{is\_first\_available}(\gamma'_a)\) is true. Since no \(\gamma\) can succeed \(\gamma'_a\) in recent context, then \(\text{pot\_init}(\gamma) \triangleq \gamma = \gamma'_a\). In other words, \(\text{pot\_init}\) only holds true for the event occurrence equivalent to \(\gamma'_a\) that is the most recent event occurrence. In case (ii), \(\text{pot\_init}(\gamma) \triangleq \text{false}\) since \(\text{cur}(E, E', E'')\) is empty. Therefore, the connector set consists of only the most recent event occurrence or it is empty. N.B. case (ii) only occurs if there are terminator occurrences signalled without any preceding unexpired initiator event occurrences. \textbf{Q.E.D.}

**Lemma 7.5**

Connector set invariant for implicit invocation holds for chronicle event context:

In the chronicle event context, \(\mathcal{Q}_E(E)\) contains the earliest available occurrence and its successors of \(E'\) if \(E' \in \alpha(E)\).
Proof: By using Def. 6.34 on p. 127 and letting $X = \text{chronicle}$, $\gamma_\alpha = \gamma^a$, $\mathcal{E}_\alpha = \{E\}'$, $\mathcal{E}_\omega = \{E\}''$, and $\gamma_\omega = \gamma^b$ then Eq. 7.1 on p. 156 can be rewritten as Eq. 7.7.

$$\text{cur}(E, E', E'') =$$
$$\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID}}' \} \land$$
$$\exists \gamma^b \in \mathcal{V}_{\text{ALID}}''$$
$$(\text{chronicle}_\alpha(E, \gamma^a, \{E\}')) \land$$
$$(\text{chronicle}_\omega(E, \gamma^b, \{E\}'')) \land$$
$$(\text{chronicle}_\alpha\omega(E, \gamma^a, \gamma^b, \{E\}', \{E\}''))$$

By replacing the different parts of \text{chronicle} in Def. 6.39 on p. 129, then Eq. 7.8 is obtained.

$$\text{cur}(E, E', E'') =$$
$$\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID}}' \} \land$$
$$\exists \gamma^b \in \mathcal{V}_{\text{ALID}}''$$
$$(\forall \gamma \in \mathcal{U}_{\text{CONS}_\alpha} (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \land \gamma^a \in \mathcal{U}_{\text{CONS}_\alpha} \land$$
$$\forall \gamma \in \mathcal{U}_{\text{CONS}_\omega} (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma) \land \gamma^b \in \mathcal{U}_{\text{CONS}_\omega} \land$$
$$\text{true})}$$

Since Lemma 7.1 on p. 156 and Lemma 7.2 on p. 156 holds, then the sets $\mathcal{U}_{\text{CONS}_\alpha}$ and $\mathcal{U}_{\text{CONS}_\omega}$ can be substituted with $\mathcal{U}_{\text{CONS}'}$ and $\mathcal{U}_{\text{CONS}''}$. Additionally, the $\text{true}$ can be removed. These substitutions give Eq. 7.9.

$$\text{cur}(E, E', E'') =$$
$$\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID}}' \} \land$$
$$\exists \gamma^b \in \mathcal{V}_{\text{ALID}}''$$
$$(\forall \gamma \in \mathcal{U}_{\text{CONS}'} (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \land \gamma^a \in \mathcal{U}_{\text{CONS}'} \land$$
$$\forall \gamma \in \mathcal{U}_{\text{CONS}''} (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma) \land \gamma^b \in \mathcal{U}_{\text{CONS}''})}$$

Since $\gamma^b$ does not occur in the subpredicate $\forall \gamma \in \mathcal{U}_{\text{CONS}'} (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \land \gamma^a \in \mathcal{U}_{\text{CONS}'}$ and Lemma 7.3 on p. 157 holds, then Eq. 7.10 on the facing page can be obtained.
cur\((E, E', E'')\) =
\[
\{ \gamma^a | \gamma^a \in VALID' \land \forall \gamma \in UNCONS' (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \land \\
\gamma^a \in UNCONS' \land \\
\exists \gamma^b \in VALID'' \\
(\forall \gamma \in UNCONS'' (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma)) \}
\] (7.10)

Since \(UNCONS' \subseteq VALID'\) and \(UNCONS'' \subseteq VALID''\), then Eq. 7.11 is obtained.

\[
\text{cur}\((E, E', E'')\) = \\
\{ \gamma^a | \gamma^a \in UNCONS' \land \forall \gamma \in UNCONS' (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \land \\
\exists \gamma^b \in UNCONS'' \\
(\forall \gamma \in UNCONS'' (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma)) \}
\] (7.11)

Eq. 7.11 can be reduced to Eq. 7.12, since there is no relation between the subpredicate \(\gamma^b \in UNCONS'' (\forall \gamma \in UNCONS'' (\gamma^b \neq \gamma \Rightarrow \gamma^b \prec \gamma))\) and \(\gamma^a\).

\[
\text{cur}\((E, E', E'')\) = \\
\{ \gamma^a | \gamma^a \in UNCONS' \land \forall \gamma \in UNCONS' (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \}
\] (7.12)

In other words, cur\((E, E', E'')\) contains only the earliest available occurrence of \(E'\). By letting \(\gamma^a = \gamma'_a \in UNCONS'\) be the earliest available initiator occurrence, then cur\((E, E', E'')\) = \(\{ \gamma'_a \}\). The predicate pot_init can be rewritten as: \(\text{pot_init}(\gamma) \iff \exists \gamma^c \in \text{cur}(E, E', E'') (\text{is_first_available}(\gamma^c) \land (\gamma = \gamma^c \lor \gamma > \gamma^c))\). This can be reduced to: \(\text{pot_init}(\gamma) \iff \gamma = \gamma'_a \lor \gamma > \gamma'_a\). In other words, pot_init holds true for event occurrences equivalent to \(\gamma'_a\) and its successors. If \(UNCONS' = \emptyset\), then the connector set is empty. Q.E.D.

**Lemma 7.6**
Connector set invariant for implicit invocation holds for continuous event context:
In the continuous event context, \(Q_E(E)\) contains the earliest available occurrence and its successors of \(E'\) if \(E' \in \alpha(E)\).
Proof: By using Def. 6.34 on p. 127 and letting $\gamma_\alpha = \gamma^a$, $\mathcal{E}_\alpha = \{E\}'$, $\mathcal{E}_\omega = \{E''\}$, and $\gamma_\omega = \gamma^b$ then Eq. 7.1 on p. 156 can be rewritten as Eq. 7.13.

$$\text{cur}(E, E', E'') = \left\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID}'} \land \exists \gamma^b \in \mathcal{V}_{\text{ALID}''} \left( \begin{array}{l}
\text{continuous}_{\alpha}(E, \gamma^a, \{E\}') \land \\
\text{continuous}_{\omega}(E, \gamma^b, \{E''\}) \land \\
\text{continuous}_{\alpha \omega}(E, \gamma^a, \gamma^b, \{E\}', \{E''\})
\end{array} \right) \right\}$$  (7.13)

By replacing the different parts of \textit{continuous} in Def. 6.41 on p. 130, Eq. 7.14 is obtained.

$$\text{cur}(E, E', E'') = \left\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID}'} \land \exists \gamma^b \in \mathcal{V}_{\text{ALID}''} \left( \begin{array}{l}
(\forall \gamma \in \mathcal{U}_{\text{NCONS}}_{\alpha} (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \land \gamma^a \in \mathcal{U}_{\text{NCONS}}_{\alpha} \land \\
\gamma^b \in \mathcal{V}_{\text{ALID}_\omega} \land \\
\neg \exists \gamma \in \mathcal{V}_{\text{ALID}_\omega} (\gamma^b \neq \gamma \land \gamma^a \prec \gamma \prec \gamma^b)
\end{array} \right) \right\}$$  (7.14)

Since Lemma 7.1 on p. 156 and Lemma 7.2 on p. 156 holds, then $\mathcal{V}_{\text{ALID}_\alpha}$ and $\mathcal{V}_{\text{ALID}_\omega}$ can be substituted with $\mathcal{V}_{\text{ALID}'}$ and $\mathcal{V}_{\text{ALID}''}$. Likewise, the sets $\mathcal{U}_{\text{NCONS}}_{\alpha}$ and $\mathcal{U}_{\text{NCONS}}_{\omega}$ can be replaced with $\mathcal{U}_{\text{NCONS}'}$ and $\mathcal{U}_{\text{NCONS}''}$. These substitutions give Eq. 7.15.

$$\text{cur}(E, E', E'') = \left\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID}'} \land \exists \gamma^b \in \mathcal{V}_{\text{ALID}''} \left( \begin{array}{l}
(\forall \gamma \in \mathcal{U}_{\text{NCONS}}' (\gamma^a \neq \gamma \Rightarrow \gamma^a \prec \gamma) \land \gamma^a \in \mathcal{U}_{\text{NCONS}}' \land \\
\gamma^b \in \mathcal{V}_{\text{ALID}''} \land \\
\neg \exists \gamma \in \mathcal{V}_{\text{ALID}''} (\gamma^b \neq \gamma \land \gamma^a \prec \gamma \prec \gamma^b)
\end{array} \right) \right\}$$  (7.15)
Since $\gamma^b$ does not occur in the subpredicate $\forall \gamma \in UNCONS' (\gamma^a \neq \gamma \Rightarrow \gamma^a < \gamma) \land \gamma^a \in UNCONS'$ and Lemma 7.3 on p. 157 holds, then Eq. 7.16 can be obtained.

\[
\text{cur}(E, E', E'') = \\
\{ \gamma^a \mid \gamma^a \in VALID' \land \forall \gamma \in UNCONS' (\gamma^a \neq \gamma \Rightarrow \gamma^a < \gamma) \land \\
\exists \gamma^b \in VALID'' \\
(\gamma^b \in VALID'' \land \\
\neg \exists \gamma \in VALID'' (\gamma^b \neq \gamma \land \gamma^a < \gamma < \gamma^b)) \}
\]

(7.16)

Since $\text{cur}(E, E', E'')$ contains only the earliest available occurrence of $E'$. This is exactly the same set that is returned for $\text{cur}(E, E', E'')$ in chronicle event context for the connector set invariant and thus the same end proof of Lemma 7.5 on p. 159 holds here.

Q.E.D.

Lemma 7.7
Connector set invariant for implicit invocation holds for general event context:
In general event context, $Q_E(E)$ contains all initiators of $E'$ that have not expired if $E' \in \alpha(E)$. 
Proof: By using Def. 6.34 on p. 127 and letting $\gamma_\alpha = \gamma^a$, $\mathcal{E}_\alpha = \{E'\}$, $\mathcal{E}_\omega = \{E''\}$, and $\gamma_\omega = \gamma^b$ then Eq. 7.1 on p. 156 can be rewritten as Eq. 7.19.

\[
\text{cur}(E, E', E'') =
\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID'}} \land
\exists \gamma^b \in \mathcal{V}_{\text{ALID''}}
\begin{align*}
& \text{general}_\alpha(E, \gamma^a, \{E'\}) \land \\
& \text{general}_\omega(E, \gamma^b, \{E''\}) \land \\
& \text{general}_\alpha\omega(E, \gamma^a, \gamma^b, \{E', E''\}) \}
\] (7.19)

By replacing the different parts of \(\text{general}\) in Def. 6.42 on p. 130, the equation Eq. 7.20 is obtained.

\[
\text{cur}(E, E', E'') =
\{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID'}} \land
\exists \gamma^b \in \mathcal{V}_{\text{ALID''}}
\begin{align*}
& (\gamma^a \in \mathcal{V}_{\text{ALID'}}) \land \\
& (\gamma^b \in \mathcal{V}_{\text{ALID''}}) \land \\
& \text{general combination not used before}(E, \gamma^a, \gamma^b, \mathcal{E}_\alpha, \mathcal{E}_\omega) \land \\
& (\mathcal{E}_\alpha \neq \mathcal{E}_\omega \Rightarrow \text{all prec. occs. processed}_1(E, \gamma^a, \gamma^b, \mathcal{E}_\alpha, \mathcal{E}_\omega)) \land \\
& (\mathcal{E}_\alpha = \mathcal{E}_\omega \Rightarrow \text{all prec. occs. processed}_2(E, \gamma^a, \gamma^b, \mathcal{E}_\alpha, \mathcal{E}_\omega)) \}
\] (7.20)

Since the dummy terminator occurrence $\Gamma(E'', [\infty, \infty]) \in G$ and it is never used for any event composition, then $\exists \gamma^b \in \mathcal{V}_{\text{ALID''}}(\ldots)$ can be replaced with true. The reason is that this part of the predicate only relates to the event occurrences that have been combined to form composite event occurrences. Therefore, Eq. 7.20 can be rewritten as Eq. 7.21.

\[
\text{cur}(E, E', E'') = \{ \gamma^a | \gamma^a \in \mathcal{V}_{\text{ALID'}} \}
\] (7.21)

In other words, $\text{cur}(E, E', E'')$ contains every initiator that has not expired nor been invalidated. The \textit{is first available} predicate is true for the first event occurrence in $\text{cur}(E, E', E'')$. The \textit{pot init} predicate is true for all event occurrences in $\text{cur}(E, E', E'')$. Therefore, all connector sets containing initiators are equal to $\text{cur}(E, E', E'')$. 

Theorem 7.1
Any context predicate that restricts an initiator occurrence and a terminator occurrence with respect to $\alpha(E)$ and $\omega(E)$ and with respect to expiration, invalidation, and reusability can be used in the connector set invariant for implicit invocation.

Proof: Any such event context is a variation of recent, chronicle, continuous, or general event context for which Lemma 7.4 on p. 157 to Lemma 7.7 on p. 163 holds. Consider Table 7.1 derived from the event context predicates, where “relative” denotes that the initiator or terminator is constrained relative to the other and “absolute” denotes that the initiator or terminator is constrained with respect to the history of processing of its (contributing) event type only. No combination of basic absolute or relative ordering constraints on initiators and terminators is left out of the table and, therefore, the existing predicates define all possible combinations. Thus, variations of the event contexts such as $\text{recent}^n$, which saves the $n$ most recent event occurrences, can be defined as a variation of the existing recent event context.

<table>
<thead>
<tr>
<th>Event context</th>
<th>Initiator</th>
<th>Terminator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recent</td>
<td>Relative</td>
<td>Absolute</td>
</tr>
<tr>
<td>Chronicle</td>
<td>Absolute</td>
<td>Absolute</td>
</tr>
<tr>
<td>Continuous</td>
<td>Absolute</td>
<td>Relative</td>
</tr>
<tr>
<td>General</td>
<td>Relative</td>
<td>Relative</td>
</tr>
</tbody>
</table>

Table 7.1: Restriction category of initiators and terminator occurrences

7.1.2 Example of Usage of Invariant
Consider the realization of the event operator sequence in Fig. 6.1 on p. 99 where $\text{expire}(E)=\infty$. To prove that the connector set invariant for implicit invocation (in Def. 7.2 on p. 154) holds, Hoare logic can be employed [Hoa69] (inference rules can be found in Section C on p. 369). The point in the execution where the invariant must hold is before the $\text{receive}$ statement, since the processing is atomic and made in isolation of other processes. A complete proof outline is depicted in Fig. 7.1 on p. 176. An additional “else” statement is added to enable a complete proof outline.
To perform a proof, the invariants in Appendix C are used to check that each triple \( \{\{P\}\} \) \( \mathcal{S} \) \( \{\{Q\}\} \) holds. For example, that \( \{\{I_{P1} \land (Q1 = Q_{E1}(E) \setminus \{e\})\} \mathcal{Q}_{E1}(E) \setminus \{e\}\} \) \( Q1 := Q1 \cup \{e\} \) \( \{\{I_{P1} \land Q1 = Q_{E1}(E)\}\} \) holds by using the assignment axiom\(^1\). If all these triplets hold, then it is necessary to demonstrate that the end state of the loop implies the start state. That is, \( I_{P} \land Q1 = Q_{E1}(E) \land Q_{E1}(E) = \emptyset \Rightarrow I_{C} \land Q1 = Q_{E1}(E) \land Q_{E1}(E) = \emptyset \) (cf. Fig. 7.1 on p. 176).

### 7.1.3 Invariant for Explicit Invocation

The invariant for explicit invocation is based on the invariant for implicit invocation. Therefore, only a proof outline is given for this invariant.

**Definition 7.4**

**Connector set invariant for explicit invocation:** In addition to the assumption in Def. 7.2 on p. 154, let \( \gamma_{\alpha}^{l} \) be the last processed initiator occurrence and, likewise, \( \gamma_{\omega}^{l} \) be the last processed terminator occurrence. Moreover,

\[
\text{succeeds}_{\text{processed initiator}}(\gamma) \overset{\Delta}{=} \text{type}(\gamma) \in \mathcal{V}_{\text{ALID}}' \land \gamma > \gamma_{\alpha}^{l}
\]

\[
\text{succeeds}_{\text{processed terminator}}(\gamma) \overset{\Delta}{=} \text{type}(\gamma) \in \mathcal{V}_{\text{ALID}}'' \land \gamma > \gamma_{\omega}^{l}
\]

then:

\[
\forall E, E', E'' (E' \in \alpha(E) \land E'' \in \omega(E) \land E' \neq E'' \Rightarrow \quad Q_{E}(E) = \{\gamma' | \text{pot}_{\text{init}}(\gamma') \lor \text{succeeds}_{\text{processed initiator}}(\gamma')\} \land \\
\forall E, E' (E' \in \omega(E) \land E' \notin \alpha(E)) \Rightarrow \quad Q_{E}(E) = \{\gamma | \text{succeeds}_{\text{processed terminator}}(\gamma)\})
\]

\(\square\)

In other words, a connector set between a contributing event type that can be an initiator and the contributed event type is defined by the context

---

\(^1\)In proof outlines, the substitution of the assignment axiom are not necessary (e.g., Hoare [Hoa69], Andrews [And91]).
7.2 Scheduling of Evaluation of Event Composition

7.2 Scheduling of Evaluation of Event Composition

Evaluation of operator rules is a complex issue, since it depends on the past processing of event monitoring as well as the event type specifications. There are two scheduling policies: data-driven and control-driven. The data-driven scheduling of composition is the result of the recursive call (such as an implementation of Snoop signalling), where the last composed event occurrence is used for the next evaluation of rule. This policy can introduce arbitrary decisions. This works for some event operators, but not for all which will be discussed in the following paragraphs. In contrast, control-driven scheduling is proposed as a means of removing arbitrary decisions (concerning what operator rule to evaluate).

To present the scheduling policies, an example is discussed before the general procedures are defined. Assume that $E_a = (E_1; E_2) \triangle E_3$ and $E_b = E_2 \triangle E_3$ are monitored and $G = \{\Gamma(E_1, [0, 0])\}$ at time 0. Then, at time 1 both an \( \Gamma(E_2, [1, 1]) \) and an \( \Gamma(E_3, [1, 1]) \) is delivered. That is, the event history is $G' = \{\Gamma(E_1, [0, 0]), \Gamma(E_2, [1, 1]), \Gamma(E_3, [1, 1])\}$ at time 1 before event composition has taken place. In data-driven scheduling following Snoop, the order of generation may be either $\Gamma(E_b, [0, 1])$, $\Gamma(E_1; E_2, [0, 1])$, and $\Gamma(E_a, [0, 1])$ or $\Gamma(E_1; E_2, [0, 1])$, $\Gamma(E_a, [0, 1])$, and $\Gamma(E_b, [0, 1])$. In other words, data-driven scheduling (in Snoop) the next rule to be evaluated is based on the last resulting event occurrence. If there are no more rules to be evaluated, back-
track to the last result and continue evaluating rules. In contrast, a control-driven scheduling requires that all (rules of) contributing event types of a composite event type are evaluated before (the rules of) the composite event type. In this example, using a control-driven scheduling algorithm, first $\Gamma(E_1; E_2, [0,1])$ and $\Gamma(E_b, [0,1])$ are added to $G$ before $\Gamma(E_a, [0,1])$ is added.

The major problem of data-driven scheduling is the treatment of the disjunctive event operator (and similar event operators). Assume that we have $E_d = E_1 \lor E_2$ and $G = \{\Gamma(E_1, [0,1]), \Gamma(E_2, [1,1])\}$. Data-Driven scheduling will add $\Gamma(E_1 \lor E_2, [0,1])$ and $\Gamma(E_1 \lor E_2, [1,1])$ to $G$ in an arbitrary order. If simultaneously terminating event occurrences of contributing event types are required to contribute to one composite event occurrence, then the correct result of the evaluation should only be $\Gamma(E_1 \lor E_2, [0,1])$ (where both $\Gamma(E_1, [0,1])$ and $\Gamma(E_2, [1,1])$ are contributing). Regardless of the policy for simultaneously terminating occurrences of the contributing event types ($E_1$ and $E_2$) of an event type ($E_d$) whose principal operator is disjunction, data-driven scheduling may (or even will\(^2\)) end up with an incorrect result.

7.2.1 Data-driven Scheduling

To define the data driven scheduling based on the Def. 6.11 on p. 109, we have chosen to transform the recursive procedure(s) of Snoop into the iteration (in the definition) by manually adding a stack that contains composed event occurrences. Let $S = \{(i, \gamma)\}$ be a stack of event occurrences. The following axiom holds for stack of event occurrences:

**Axiom 7.1**

Stack of event occurrence axioms:

\[
\forall S, i, \gamma ((i, \gamma) \in S \Rightarrow i \geq 1) \quad (a)
\]

\[
\forall S (|S| = 1 \Rightarrow \forall i, \gamma ((i, \gamma) \in S \land i = 1)) \quad (b)
\]

\[
\forall S, i, \gamma, j, \gamma' ((i, \gamma), (j, \gamma') \in S \land \gamma \neq \gamma' \Rightarrow i \neq j) \quad (c)
\]

\[\square\]

\(^2\)If the resulting occurrence should be aggregated from the simultaneously terminating event occurrences, then it will be incorrect.
In other words: (a) the index of a stack is a natural number greater than or equal to 1; (b) if there is only one element in a stack, then its order index is 1; (c) the index is unique (i.e., no two distinct event occurrences on the stack may have the same index).

**Definition 7.5**

Basic stack operations:

\[ i = \max \text{order}(S') \text{ where} \]
\[ (\forall j, \gamma ((j, \gamma) \in S' \Rightarrow i \geq j) \land \exists j, \gamma ((j, \gamma) \in S' \land i = j)) \lor (|S'| = 0 \Rightarrow i = 0) \]  

(a)

\[ S = \text{push}(S', \gamma) \text{ where} \]
\[ S = S' \cup \{ (\max \text{order}(S') + 1, \gamma) \} \]  

(b)

\[ S = \text{pop}(S') \text{ where} \]
\[ S = S' \setminus \{ (i, \gamma) | i = \max \text{order}(S') \land (i, \gamma) \in S' \} \]  

(c)

\[ \gamma = \text{top}(S') \text{ where} \]
\[ \exists i, \gamma' (i = \max \text{order}(S')(i, \gamma') \in S' \land \gamma = \gamma') \]  

(c)

☐

In other words, (a) the \( \max \text{order} \) function returns the maximum index of the stack; (b) the push function returns a stack with the event occurrence pushed on top of the stack; (c) the pop function returns a stack where the top-most element of the stack has been removed; and, finally, (d) the top function returns the top-most elements (event occurrence) of the stack.

**Definition 7.6**

Contributing event occurrences function:

Let \( s \) be a state, \( r \) be a rule, and \( E \) be an event type such that \( r \in \text{rules}(E) \):

- For the usage rule form, the definition is:

\[ \text{contributors}(s, r, E) = \{ \gamma_1, \ldots, \gamma_n | \gamma_1, \ldots, \gamma_n \in G (R \land R_c \land H) \} \]
For invalidation rule form, the definition is:

\[
\text{contributors}(s, r, E) = \\
\{ \gamma_i, \ldots, \gamma_m | \gamma_i, \ldots, \gamma_m \in \mathcal{G}(R_{im} \land \forall \gamma_j, \ldots, \gamma_n \in \mathcal{G}(R_{jn} \land R_c \Rightarrow H')) \}
\]

□

Definition 7.7

Data driven scheduling of event composition (from Snoop): This definition is based on the algorithm in Def. 6.11 on p. 109. To recapitulate, let \( s \) be the state of event composition; let \( I = (later\_state(s, s_0) \lor s=s_0) \), and let \( C = r \in \text{rules}(E) \land \text{condition}(s, r, E) \). Additionally, let \( S \) be a stack of event occurrences, let \( I' = I \land (|S| > 0 \Rightarrow \text{top}(S) \in \mathcal{G}) \) and let \( C' = C \land \text{top}(S) \in \text{contributors}(s, r, E) \) then

\[
\{ \{ I' \land |S|=1 \land \text{prim}(\text{type}(\text{top}(S))) \} \}
\]

while \(|S| > 0 \) do begin

\[
\{ \{ I' \land |S| > 0 \} \}
\]

while \( \exists E, r (C') \) do begin

\[
\{ \{ I \land |S| > 0 \land E, r (C') \} \}
\]

select \( E, r \) such that \( C' \);

\[
\{ \{ I \land |S| > 0 \land C' \} \}
\]

\( \mathcal{G}_{old} := \mathcal{G} \);

\( s := \text{conclusion}(s, r, E); /\ast \text{updates state } s */ \)

\[
\{ \{ I \land |S| > 0 \land later\_state(s, s') \land \mathcal{G}_{old} \subseteq \mathcal{G} \} \}
\]

if \( \mathcal{G} \setminus \mathcal{G}_{old} \neq \emptyset \) then

\[
\{ \{ I \land |S| > 0 \land later\_state(s, s') \land \mathcal{G}_{old} \subseteq \mathcal{G} \land \text{top}(S) \notin (\mathcal{G} \setminus \mathcal{G}_{old}) \} \}
\]

foreach \( e \) in \( \mathcal{G} \setminus \mathcal{G}_{old} \) do begin

\( S := \text{push}(S, e); \)

end

endif

\[
\{ \{ I \land |S| > 0 \land later\_state(s, s') \land (\mathcal{G} \setminus \mathcal{G}_{old} \neq \emptyset \Rightarrow \text{top}(S) \in (\mathcal{G} \setminus \mathcal{G}_{old})) \} \}
\]

end

\[
\{ \{ I \land |S| > 0 \land \neg \exists E, r(C') \} \}
\]

\( S := \text{pop}(S); \)

\[
\{ \{ I \land |S| = 0 \} \}
\]

end

\[
\{ \{ I \land \neg \exists E, r(C') \land |S| = 0 \} \}
\]

□
In other words, we know that the first event occurrence is primitive (immediate invocation). The stack is used to remember the last composed event occurrences and the initiating primitive event occurrence. The algorithm in Def. 6.11 on p. 109 is augmented with this stack to simulate recursive calls. Any selected event type and rule must use the last composed event occurrence or the initiating primitive event occurrence to be eligible. When there are no more eligible event types and rules left, pop the top element of the stack. The algorithm is completed when there are no more event occurrences on the stack.

### 7.2.2 Control-Driven Scheduling of Event Composition

One way to realize a control-driven scheduling policy is to assign an order to each event type in which it should be executed. Then, this order is used to invoke event composition for the event types in order. Thereby, it is possible to ensure correct results, since all contributing event types are guaranteed to be evaluated before a contributed event type. Let $s_{\text{ord}}(E)$ be the order in which evaluation of $E$ should be scheduled with respect to all other event types.

**Definition 7.8**

The order of scheduling:

$$\forall E, E' (E' \in \text{alloperands}(E) \Rightarrow s_{\text{ord}}(E') < s_{\text{ord}}(E))$$

□

In other words, a contributing event type $E'$ should have a lower order than the composite event type $E$ and, hence, rules of $E'$ should be scheduled for composition before rules of $E$.

**Definition 7.9**

Scheduling order of primitive event types:

$$\forall E (\text{prim}(E) \Rightarrow s_{\text{ord}}(E)=0)$$

□
In other words, the scheduling order of primitive event types is 0 (meaning that they are evaluated first).

**Definition 7.10**

**Maximum order:**

\[ s_{ord_{max}} \in \mathbb{N} \land \forall E (\exists E' (E' \in \text{all operands}(E) \land s_{ord_{max}} = s_{ord}(E')) \land \forall E'' (E'' \in \text{all operands}(E) \land E' \neq E'' \Rightarrow s_{ord_{max}} \geq s_{ord}(E''))) \]

□

In other words, \( s_{ord_{max}} \) is the maximum order among all event types.

**Definition 7.11**

**Minimum order:**

\[ s_{ord_{min}} \in \mathbb{N} \land \forall E (\exists E' (E' \in \text{all operands}(E) \land s_{ord_{min}} = s_{ord}(E')) \land \forall E'' (E'' \in \text{all operands}(E) \land E' \neq E'' \Rightarrow s_{ord_{min}} \leq s_{ord}(E''))) \]

□

In other words, \( s_{ord_{min}} \) is the minimum order among all event types.

**Definition 7.12**

**Schedulable event type sets of order:**

\[ \text{sched\_event\_types}(i) = \begin{cases} \{E| s_{ord}(E) = i\} & \text{iff } s_{ord_{min}} \leq i \leq s_{ord_{max}} \\ \emptyset & \text{iff } \neg (s_{ord_{min}} \leq i \leq s_{ord_{max}}) \end{cases} \]

□
In other words, \texttt{s\_ord} returns a set of event types of scheduling order \(i\). If \(i\) is out of range, then an empty set is returned.

\textbf{Definition 7.13}  
\textbf{Control-driven event composition:} To recapitulate the assumptions in Def. 6.11 on p. 109, let \(s\) be the state of event composition set to \(s_i\) each iteration; let \(I = (\text{later\_state}(s, s_0) \lor s = s_0)\), and let \(C = r \in \text{rules}(E) \land \text{condition}(s, r, E)\). Additionally, let \(k\) be the currently evaluated order and let \(C'' = C \land E \in \text{sched\_event\_types}(k)\) then

\[
\begin{align*}
\text{k := s\_ord}_{\text{min}}; \\
\{\{I \land s\_ord_{\text{min}} = k\}\} \\
\text{while } k \leq s\_ord_{\text{max}} \text{ do begin} \\
\{\{I \land s\_ord_{\text{min}} \leq k \leq s\_ord_{\text{max}}\}\} \\
\text{while } \exists E, r (C'') \text{ do begin} \\
\{\{I \land s\_ord_{\text{min}} \leq k \leq s\_ord_{\text{max}} \land \exists E, r (C'')\}\} \\
\text{select } E, r \text{ such that } C''; \\
\{\{I \land s\_ord_{\text{min}} \leq k \leq s\_ord_{\text{max}} \land C''\}\} \\
\text{s := conclusion}(s, r, E); */ \text{ updates state s } */ \\
\{\{I \land s\_ord_{\text{min}} \leq k \leq s\_ord_{\text{max}} \land \text{later\_state}(s, s')\}\} */ C'' \text{ undef. } */ \\
\text{end} \\
\{\{I \land s\_ord_{\text{min}} \leq k \leq s\_ord_{\text{max}} \land \neg \exists E, r(C'')\}\} \\
k := k + 1; \\
\{\{I \land s\_ord_{\text{min}} < k \leq s\_ord_{\text{max}} + 1\}\} \\
\text{end} \\
\{\{I \land k > s\_ord_{\text{max}} \land \neg \exists E, r(C'')\}\}
\end{align*}
\]

\(\square\)

In other words, complete event composition for all event types of the same scheduling order before moving to event types of a successive scheduling order. Given this control-driven scheduling of event composition, composition will work for any event operator, since it is guaranteed that all contributing event types complete their composition before the evaluation of an event type using the results of these contributing event types.
7.3 Discussion

This chapter demonstrates two of the strong points of our schema. First of all, it is possible to derive invariants. We have demonstrated that these invariants can be used in a complete proof outline of an implementation of an event operator in an event context. Secondly, it is possible to define scheduling of event composition to, for example, achieve correct results.

To define invariants for connector sets between non-initiating and non-terminating contributing event types and the contributed event type, the semantics of the event operators must be considered. For example, in the non-occurrence event operator $E = N(E_1, E_2, E_3)$ there is nothing but the general invariant (in Def. 7.1 on p. 152) for $Q_{E_2}(E)$. This invariant does only state the type of all event occurrences in the connector set. It is considered future work to state such invariants that hold for any event operator semantics.

The issue of splitting event composition into tasks has been left out of this chapter, since it is discussed in the architectural issues in Part III.

If event occurrences arrive out of order, then the basis for the implementation strategy in this section falls. In this case, time constraints must be applied to check validity of contributing event occurrences. This issue is discussed in Section 15.3.12 on p. 317.

7.4 Related Work on Realization

In pragmatic approaches found in active databases there are no formal rules for transforming specifications of operators into an implementation. In contrast, this work provides definitions of invariants that need to be maintained by any algorithm performing event composition.

In the formal approaches such as monitoring of RTL, TCCS, and chronicle recognition, there are transformations of their operators into an implementation. However, these formal approaches are locked with their respective assumption about the context, whereas it is possible to switch between different event context in our approach.

In SMILE [Jae97], they have addressed scheduling of event composition. In contrast to our approach, their approach is less general since it is specific to the event specification language employed in SMILE. Since our control-driven scheduling is defined in terms of all operands it holds for any event specification language defined in our schema.
7.5 Summary

An advantage of the SMILE approach is that it allows hybrid scheduling, a mixture of data-driven and control-driven policies, applicable to distributed systems.

7.5 Summary

In this chapter, invariants for the connector sets have been derived from the event context predicates. Also, a control-driven scheduling for correct event composition has been defined.
Let $I_C$ be the connector set invariant, let $I_P \overset{\triangle}{=} I_C \wedge e \in G$, let $I_{P1} \overset{\triangle}{=} I_P \wedge \text{type}(e) = E_1$, let $I_{P2} \overset{\triangle}{=} I_P \wedge \text{type}(e) = E_2$, and let $\text{first}(\gamma) \overset{\triangle}{=} \gamma \in Q_{E_1}(E) \wedge \forall \gamma' \in Q_{E_1}(E) \left( \gamma \neq \gamma' \Rightarrow \gamma \prec \gamma' \right)$ then

\[
\text{while true do}
\]

\[
\{ \{ I_C \wedge Q_1 = Q_{E_1}(E) \wedge Q_{E_2}(E) = \emptyset \} \}
\]

\[
\text{receive } e \text{ where } \text{type}(e) \in \{ E_1, E_2 \} /* \text{receive event occurrence} */
\]

\[
\{ \{ I_P \wedge ((\text{type}(e) = E_1 \wedge Q_1 = Q_{E_1}(E)) \setminus \{ e \} \vee
\]

\[
(\text{type}(e) = E_2 \wedge Q_1 = Q_{E_1}(E)) \wedge Q_{E_2}(E) = \{ e \} ) \} \}
\]

\[
\text{if } \text{type}(e) = E_1 \text{ then}
\]

\[
\{ \{ I_{P1} \wedge Q_1 = Q_{E_1}(E) \} \}
\]

\[
Q_1 := Q_1 \cup \{ e \} /* \text{add } E_1 \text{ occurrence} */
\]

\[
\{ \{ I_{P1} \wedge Q_1 = Q_{E_1}(E) \} \}
\]

\[
\text{else } /* \text{must be } E_2 */
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) \wedge Q_{E_2}(E) = \{ e \} \} \}
\]

\[
\text{if not empty}(Q_1) /* \text{there is an } E_1 \text{ occurrence} */
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) \neq \emptyset \wedge Q_{E_2}(E) = \{ e \} \} \}
\]

\[
etmp := \text{earliest}(\text{sortAscendingOnTerminationTime}(Q_1));
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) \neq \emptyset \wedge Q_{E_2}(E) = \{ e \} \wedge \text{first}(etmp) \} \}
\]

\[
cev := \text{combine}(etmp, e); /* \text{compute composite event occurrence} */
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) \wedge Q_{E_2}(E) = \emptyset \wedge \text{first}(etmp) \wedge
\]

\[
\langle etmp, cev \rangle \in \mathcal{U}_e(E) \wedge \langle e, cev \rangle \in \mathcal{U}_e(E) \}
\]

\[
Q_1 := Q_1 \setminus \{ etmp \}; /* \text{consume selected } E_1 \text{ occurrence} */
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) \wedge Q_{E_2}(E) = \emptyset \wedge \neg \text{first}(etmp) \wedge
\]

\[
\langle etmp, cev \rangle \in \mathcal{U}_e(E) \wedge \langle e, cev \rangle \in \mathcal{U}_e(E) \}
\]

\[
\text{post EventForSignalling}(cev);
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) \wedge Q_{E_2}(E) = \emptyset \wedge \neg \text{first}(etmp) \wedge
\]

\[
\langle etmp, cev \rangle \in \mathcal{U}_e(E) \wedge \langle e, cev \rangle \in \mathcal{U}_e(E) \wedge cev \in G \}
\]

\[
\text{else } /* \text{No } E_1 \text{ occurrence, added to proof outline} */
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) = \emptyset \wedge Q_{E_2}(E) = \{ e \} \} \}
\]

\[
\mathcal{I}_e(E) = \mathcal{I}_e(E) \cup \{ e \}; /* \text{assignment of auxiliary variable} */
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) = \emptyset \wedge Q_{E_2}(E) = \emptyset \wedge e \in \mathcal{I}_e(E) \} \}
\]

\[
\text{endif}
\]

\[
\{ \{ I_{P2} \wedge Q_1 = Q_{E_1}(E) \wedge Q_{E_2}(E) = \emptyset \} \}
\]

\[
\text{endif}
\]

\[
\{ \{ I_P \wedge Q_1 = Q_{E_1}(E) \wedge Q_{E_2}(E) = \emptyset \} \}
\]

\[
\text{done}
\]

Figure 7.1: Proof outline of the sequence operator algorithm
Chapter 8

Bounded Event Composition

“I wish I had the ability to find the truth as easily as I can reveal falsehood”

/Cicero

This chapter presents a proof that the number of computational steps of event composition is bounded; this is a necessary requirement for resource predictability (cf. timeliness in Section 3.1.1 on p. 27). This proof holds for any event expression in any event context following the formalized schema presented in Ch. 6. This proof is based on the following (time) constraints on event composition: each primitive event type has a minimum interarrival time between event occurrences (i.e., primitive event types are sporadic) and each composite event type is associated with an expiration time. This expiration time has the same semantics as the Chronicle deadline used in Chronicle recognition [DGG93, Dou96] (cf. Section 4.5.6 on p. 62).

An example of the problem of unbounded event composition is that the connector set in Fig. 6.1 on p. 99 may grow indefinitely, since there is no guarantee that an $E_2$ occurrence will be signalled. For chronicle context (the context that the algorithm in Fig. 6.1 on p. 99 is based on) this is not a major problem with respect to response time given that event occurrences arrive in the correct order, but it is a major problem concerning the memory requirements (of the connector sets). However for, for example, the sequence event operator in continuous event context, the processing time increases
with the size of the connector set, since a terminator occurrence generates a composite event occurrence for each unconsumed initiator occurrence.

Given the suggested time constraints, we need to ensure that expiration is detected as early as possible. The problem of early detection of expiration is similar to early detection of violation (or satisfaction) of time constraints [ML97a]; unless there is a timeout that signals the expiration, then the expiration may go undetected (for ever). Similarly to Mok et al. [ML97a], we let the algorithm set timers that generate timeouts so that composition of initiated but not terminated composite event occurrences can be stopped. This composition is stopped by removing the initiator occurrences from the connector sets associated with the composite event type.

An improvement of the algorithm in Fig. 6.1 on p. 99 to handle early detection of expiration is found in the following example: Assume that a timeout is sent as an event occurrence of type $E_T$ to the process performing event composition of an event type whose principal event operator is sequence and the context is chronicle (as in Fig. 6.1 on p. 99), then the algorithm executed by this process (in Fig. 6.1 on p. 99) can be augmented to manage the timeout as in Fig. 8.1 on the facing page. The significant augmentations are boxed in Fig. 8.1 on the facing page. Essentially, there are two augmentations: (i) set a timeout for each event occurrence added to the connector set, and (ii) when a timeout has occurred, remove expired initiator occurrences.

Given the improved algorithm in Fig. 8.1 on the facing page, we can now give the rationale behind the chosen time constraints. Assume that $E = E_1; \langle \text{expire}: T, \text{context: chronicle} \rangle E_2$ (Fig. 8.1 on the facing page is an example implementation of this specification) is monitored and that the minimum interarrival time is $T_{\text{int}}$. Then, the connector set between $E_1$ and $E$ ($Q_1$ in the example implementation) contains at most $\lfloor T/T_{\text{int}} \rfloor + 1$ event occurrences (i.e., the connector set is bounded). This holds if the processing of a timeout require only negligible resources. In other words, one event occurrence per end of interval that fits within $T$ plus one event occurrence for the start of the first interval that fits within $T$.

To prove bounded event composition, we demonstrate that two event expressions result in the same composite event occurrences. The complete formal definitions of these event expressions are presented in Section 8.2 on p. 185, since we need to present the prerequisites of these definitions first. These event expressions are: $E_a = E_x; \langle \text{expire}: T \rangle$ and $E_b$ based on $E_x$ where $\text{expire}(E_b) = \infty$ and $E_b$ contains an explicit timeout specification (of $T$ dura-
Let $E = E_1; E_2$ in

**process** sequence<$E$>

var

$Q_1$: set<$E_1$>

$e, etmp$: any_of<$E_1, E_2$>

$cev$: $E$

while true do

/* receive signalled event occurrence, subscription implicitly derived from event specification of $E$ */

receive $e$ where type($e$) $\in \{E_1, E_2, E_T\}$ /* receive signalled occurrences*/

if type($e$) = $E_1$ then

$Q_1 := Q_1 \cup \{e\}$; /* add $E_1$ occurrence */

generate_timeout_event_at($E_T, end(span(e)) + expire(E)$);

else if type($e$) = $E_2$ then

if not empty($Q_1$) /* there is an $E_1$ occurrence */

/* select earliest available $E_1$ occurrence (view set as a queue): */

$etmp := \text{earliest(sortAscendingOnTerminationTime}(Q1))$;

$cev := \text{combine}(etmp, e)$; /* compute composite event occurrence */

$Q_1 := Q1 \setminus \{etmp\}$; /* consume selected $E_1$ occurrence */

postEventForSignalling($cev$);

endif

else /* timeout: type($e$) = $E_T$ */

$Q1 := Q1 \setminus \{\gamma | \gamma \in Q1 \land end(span(\gamma)) > end(span(e))\}$;

endif

done

---

**Figure 8.1:** Sequence operator algorithm for chronicle event context with timeout management

---

This explicit timeout specification is based on the relative temporal event operator (in Def. 4.10 on p. 49). To demonstrate this, we first introduce language constructs in the next section, in which the semantics of $E_b$ can be expressed. In Section 8.2 on p. 185, we prove that event composition is bounded. In Section 8.3 on p. 190, the management of expiration times is discussed. These sections are followed by results containing a discussion, related work and a brief summary.
8.1 Prerequisites of Early Detection of Expiration

In this section, we provide prerequisites for specifying $E_b$. These prerequisites are: (i) an event operator that can be used to specify $E_b$ (including an explanation of why event operators derived from Defs 4.3 to 4.11 on pp. 47–49 cannot be used), and (ii) a transformation function that is necessary to specify that an event occurs whenever $E_x$ is initiated. The event operator is named chronicle aperiodic and cannot be specified in first-order predicate logic. Chronicle aperiodic is similar to the aperiodic event operator, but, in contrast to the aperiodic event operator, it can handle overlapping enabling intervals (cf. Def. 4.8 on p. 48).

8.1.1 Chronicle Aperiodic Operator

The chronicle aperiodic operator (written $CA(E_1, E_2, E_3)$) requires that the enabling interval of the event expression (similarly to the aperiodic event operator Def. 4.8 on p. 48, the half-open interval of between $E_1$ and $E_3$) follows the chronicle context. The enabling interval should follow chronicle context regardless of the event context of both the $E_2$ event type and the event expression whose principal operator is chronicle aperiodic. To manage this, it is necessary to keep track of the enabling interval of the event operator in addition to keeping track of used and invalidated initiator and terminator occurrences. Therefore, we introduce the predicates $ca_{ei}$ and $ca_{et}$ (read “holds true for the earliest available initiator (and terminator respectively) of the chronicle aperiodic enabling interval”). These two predicates are similar to $chronicle_{ei}$ and $chronicle_{et}$ (in Def. 6.39 on p. 129). To support the introduced predicates, we extend the formalized schema in Ch. 6 with the sets $I^4_{ei}(E)$ and $I^4_{et}(E)$ for the initiators and terminators of the enabling interval ($E_1$ and $E_3$ in $CA(E_1, E_2, E_3)$). We only use invalidation sets to keep track of the enabling intervals, since they are never directly used to form a composite event occurrence.
Definition 8.1
Chronicle aperiodic: Let $\text{UNCONS}_{\alpha}^A \triangleq \mathcal{A}C_{\alpha} \setminus I_{\alpha}^A(E)$, $\text{UNCONS}_{\omega}^A \triangleq \mathcal{A}C_{\omega} \setminus I_{\omega}^A(E)$ (where $\mathcal{A}C_{\alpha}$ and $\mathcal{A}C_{\omega}$ are defined in Def. 6.36 on p. 128, and $\text{UNCONS}_{\alpha}^A$ and $\text{UNCONS}_{\omega}^A$ are similar to $\text{UNCONS}_{\alpha}$ and $\text{UNCONS}_{\omega}$ in Def. 6.38 on p. 128) in:

\[
\text{ca}_{\alpha}(E, \gamma_{\alpha}, E_{\alpha}) \triangleq \forall \gamma \in \text{UNCONS}_{\alpha}^A (\gamma_{\alpha} \neq \gamma \Rightarrow \gamma_{\alpha} \prec \gamma) \land \gamma_{\alpha} \in \text{UNCONS}_{\alpha}^A
\]

\[
\text{ca}_{\omega}(E, \gamma_{\omega}, E_{\omega}) \triangleq \forall \gamma \in \text{UNCONS}_{\omega}^A (\gamma_{\omega} \neq \gamma \Rightarrow \gamma_{\omega} \prec \gamma) \land \gamma_{\omega} \in \text{UNCONS}_{\omega}^A
\]

Let $E \triangleq CA(E_1, E_2, E_3)$ and $\gamma \triangleq \Gamma(E, \text{span}(\gamma_2))$ in the ca-rules:

\[
\forall E \forall \gamma_1, \gamma_2 \in \mathcal{G}:
\]

\[
\text{type}(\gamma_1) = E_1 \land \text{ca}_{\alpha}(E, \gamma_1, \{E_1\}) \land \text{type}(\gamma_2) = E_2
\]

\[
X(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\})
\]

\[
\gamma_1 \prec \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2)
\]

\[
\neg \exists \gamma_3 \in \mathcal{G} \left( \text{type}(\gamma_3) = E_3 \land \gamma_3 \notin I_{\alpha}^A(E) \land \gamma_1 \prec \gamma_3 \land \neg(\gamma_1 \parallel \gamma_3) \land \gamma_2 \succ \gamma_3 \right)
\]

\[
\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_{\alpha}(E) \cup = \{\gamma_2, \gamma\} \quad \mathcal{U}_{\omega}(E) \cup = \{\gamma_2, \gamma\}
\]

\[
\text{ca}
\]

\[
\forall E \forall \gamma_2 \in \mathcal{G}:
\]

\[
\text{type}(\gamma_2) = E_2
\]

\[
X_{\omega}(E, \gamma_2, \{E_2\})
\]

\[
\forall \gamma_1 \in \mathcal{G} \left( \text{type}(\gamma_1) = E_1 \land \text{ca}_{\alpha}(E, \gamma_1, \{E_1\}) \land \right.
\]

\[
X(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\}) \Rightarrow
\]

\[
\gamma_2 \prec \gamma_1 \lor \gamma_2 \parallel \gamma_1
\]

\[
I_{\alpha}(E) \cup = \{\gamma_2\} \quad I_{\omega}(E) \cup = \{\gamma_2\}
\]

\[
\text{ca}'
\]
∀\( E \forall \gamma_1, \gamma_3 \in G : \)
\[
\text{type}(\gamma_1) = E_1 \land \text{type}(\gamma_3) = E_3
\]
\[
\text{ca}_{ei}(E, \gamma_1, \{E_1\}) \land \text{ca}_{ei}(E, \gamma_3, \{E_3\})
\]
\[
\gamma_1 \prec \gamma_3 \land \neg(\gamma_1 \parallel \gamma_3)
\]
\[
\neg \exists \gamma_2 \in G (\text{type}(\gamma_2) = E_2 \land \text{X}(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\}) \land
\]
\[
\gamma_1 \prec \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2) \land \gamma_3 \succ \gamma_2
\]
\[
\Gamma_{\alpha}^A(E) \cup = \{\gamma_1\} \quad \Gamma_{\omega}^A(E) \cup = \{\gamma_3\}
\]

∀\( E \forall \gamma_3 \in G : \)
\[
\text{type}(\gamma_3) = E_3
\]
\[
\text{ca}_{ei}(E, \gamma_3, \{E_3\})
\]
\[
\forall \gamma_1 \in G (\text{type}(\gamma_1) = E_1 \land \text{ca}_{ei}(E, \gamma_1, \{E_1\}) \Rightarrow \gamma_3 \prec \gamma_1 \lor \gamma_3 \parallel \gamma_1)
\]
\[
\Gamma_{\omega}^A(E) \cup = \{\gamma_3\}
\]

In other words for the ca-rule, if there is an earliest available occurrence of \( E_1 \) that precedes an earliest available occurrence of \( E_2 \) such that there is no non-invalidated \( E_3 \) occurrence that precedes the \( E_2 \) occurrence while succeeding the \( E_1 \) occurrence, then the chronicle aperiodic event has occurred with the same span as the \( E_2 \) occurrence. An \( E_1 \) occurrence is available if it has been generated and not invalidated by the chronicle aperiodic operator. The \textit{ca}_{ei} \text{ } predicate is not used in the ca-rule, since the \text{type}(\gamma_1) = E_3 \land \gamma_3 \notin \Gamma_{\omega}^A(E) \text{ } in \exists \gamma_3 \in G (\text{type}(\gamma_3) = E_3 \land \gamma_3 \notin \Gamma_{\omega}^A(E) \land \gamma_3 \succ \gamma_1 \land \neg(\gamma_3 \parallel \gamma_1) \land \gamma_3 \prec \gamma_2)

is more relaxed than the \textit{ca}_{ei} \text{ (E, } \gamma_r, \{E_3\}) \text{ predicate}. In contrast to the ca-rule, the a-rule of the aperiodic event operator (in Def. B.3 on p. 366) does not check which enabling intervals have been invalidated. That is, there is no maintenance of earliest available event occurrences with respect to the enabling interval in the a-rule. The precedent of the a-rule is only true for each \( E_2 \) occurrence that succeeds an \( E_1 \) occurrence and this \( E_2 \) occurrence is not preceded or overlapped by an \( E_3 \) occurrence that also succeeds the \( E_1 \) occurrence.
8.1 Prerequisites of Early Detection of Expiration

In other words for the ca’-rule, if earliest available event occurrence of $E_1$ either succeeds or overlaps with the earliest available event occurrence of $E_2$ that satisfies the context predicate, then invalidate this $E_2$ occurrence.

In other words for the ca’′-rule, if there is an earliest available $E_1$ occurrence followed by an earliest available $E_3$ occurrence that satisfies the $ca_{ei_\alpha}$ and $ca_{ei_\omega}$ predicates and there is no earliest available $E_2$ occurrence such that it succeeds or overlaps with the $E_1$ occurrence while the $E_2$ occurrence precedes the $E_3$ occurrence, then invalidate the $E_1$ and $E_3$ occurrence.

In other words for the ca’′′-rule, if the earliest available terminator occurrence of an enabling interval is not associated with any initiator occurrence, then this terminator is invalidated. The proof of the rules for the chronicle aperiodic event operator are found in Table 8.1.

<table>
<thead>
<tr>
<th>#</th>
<th>Precedence</th>
<th>Overlap</th>
<th>Rule</th>
<th>U/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(1, 2, ?)</td>
<td>F</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2.</td>
<td>(1, 2, ?)</td>
<td>T</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>4.</td>
<td>(1, -2, 3, ?)</td>
<td>?</td>
<td>?</td>
<td>F</td>
</tr>
<tr>
<td>5.</td>
<td>(1, -2, 3, ?)</td>
<td>?</td>
<td>?</td>
<td>T</td>
</tr>
<tr>
<td>6.</td>
<td>(3, ?)</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Proof table for the chronicle aperiodic operator

8.1.2 Aperiodic versus Chronicle Aperiodic Operator

The definition of aperiodic event expressions contains a problem for early detection of expiration, since it disallows overlapping enabling intervals. For example, assume that we monitor the event expressions $E$ and $E'$, where these event expressions are defined as $E = A(E_1, E_1; E_2, (E_1+10)!)$ and $E' = CA(E_1, E_1; E_2, (E_1+10)!)$; further, assume that, at time 11, $\mathcal{G} = \{\Gamma(E_1, [0,0]), \Gamma(E_1, [5,5]), \Gamma((E_1+10)!, [10,10]), \Gamma(E_2, [11,11])\}$. In this example, no event of type $E$ has occurred according to the definition of aperiodic event expressions, since the $\Gamma((E_1+10)!, [10,10])$ cancels all preceding $E_1$ occurrences. In contrast, the an $\Gamma(E', [5,11])$ has occurred given that $c_{ord}(E_1) < c_{ord}(E_1; E_2) < c_{ord}((E_1+10)!)$.

The major disadvantage of the chronicle aperiodic operator is that it, in contrast to the aperiodic event operator, cannot be expressed in pure first-
order predicate logic, since it requires consumption (realized as invalidation) of the enabling interval.

8.1.3 Initialization Transformation

To specify the explicit timeout in $E_b$, we need to be able to transform $E_x$ into an event expression that occurs every time $E_x$ is initiated. The transformation function (based on Def. 6.31 on p. 126) is:

**Definition 8.2**
Initiator transformation:

$$\overline{\alpha}(E) = \bigvee_{E' \in \hat{\alpha}(E)} E'$$

In other words, the initiator transformation is a disjunction of all contributing primitive event types that may initiate an event occurrence of event type $E$. Given this, we can now specify $\overline{\alpha}(E_x) + T$ (i.e., an event occurrence that occurs throughout the interval initiated at the same time as an $E_x$ may be initiated and terminates $T$ duration of this event occurrence).

**Lemma 8.1**
An occurrence of $\overline{\alpha}(E)$ is instantaneous.

**Proof:** Since the transformation results in a disjunction of primitive events that are instantaneous (in Axiom 6.1 on p. 97), and the occurrence of a disjunction has the same span as the constituent (in Def. 6.46 on p. 139), then the occurrence of $\overline{\alpha}(E)$ is instantaneous.  \[\square\]

**Lemma 8.2**
When an event $\gamma$ of $E$ has occurred, then an event $\gamma'$ of $E' = \overline{\alpha}(E)$ has occurred such that $\text{start}(\text{span}(\gamma)) = \text{end}(\text{span}(\gamma'))$. 
8.2 Proof of Early Detection of Expiration

Proof: Since an event $E'$ occurs every time an occurrence of $E$ is potentially initiated (represented by $\gamma$), then, when $\gamma$ is terminated, there is a $\gamma'$ of type $E'$ such that the relation $\text{start}(\text{span}(\gamma)) = \text{end}(\text{span}(\gamma'))$ holds.

Q.E.D.

8.2 Proof of Early Detection of Expiration

To be more precise, we want to prove that $E_a = E_x(\text{expire}: T)$ generates the same output as $E_b = CA(\overline{\alpha}(E_x), E_x, (\overline{\alpha}(E_x)+T)!$. To do this, we show that by symbolic manipulation the generation rule $ca$ will only generate occurrences whose duration of their time spans is less than $T$.

Given the proof table Table 8.1 on p. 183 ensures that Theorem 6.1 on p. 118 holds for the chronicle aperiodic operator, we only need to work with the $ca$-rule and none of the help rules.

Lemma 8.3

Event consumption regardless of event context:

All event operators that maintain $U_\alpha(E) = U_\omega(E)$ consume the contributing event occurrences.

Proof: Examples of operators that maintain $U_\alpha(E) = U_\omega(E)$ are disjunction (Def. 6.46 on p. 139), aperiodic (Def. B.3 on p. 366), chronicle aperiodic (Def. 8.1 on p. 181). For chronicle, recent, and continuous event contexts, there is at least one contributing event type that is consumed during event composition (cf. Theorem 7.1 on p. 165). Thus, since the contributing event occurrences of $E$ are used as both initiators and terminators, then the context predicates cannot hold true for a used event occurrence. In other words, these used event occurrences are consumed.

For general event context (Def. 6.42 on p. 130), no combination of initiator and terminator event occurrences can be used twice to form different composite event occurrences. Further, $\gamma_\alpha = \gamma_\omega$ and $E_\alpha = E_\omega$ in the event operator rules for disjunction, aperiodic, and chronicle aperiodic event operators ensuring that no two different event occurrences of the contributing event type can be used to form a composite event occurrence. This implies that $\text{all}\_\text{processed}_2$ subpredicate must be true to ensure progress. Basically, this subpredicate ensures that event occurrences of contributing
event types are processed in order. Since no combination of initiator and terminator occurrences can be reused to form a composite event occurrence, then usage implies consumption in general event context too. Q.E.D.

Lemma 8.4

No successive occurrences of initiator expression:
The precedent of the ca-rule can only hold true if the condition $\text{end}(\text{span}(\gamma_1)) \leq \text{start}(\text{span}(\gamma_2))$ holds for the candidate event occurrences $\gamma_1$ and $\gamma_2$ when the constraint $E_1 = \alpha(E_2) \land c_{\text{ord}}(E_1) < c_{\text{ord}}(E_2)$ holds.

In other words, only for event occurrences of the $E_1$ (derived from $E_2$) that precede or have the same termination time as the evaluated event occurrence of $E_2$ may the antecedent of the ca-rule hold true.

Proof: Given Lemma 8.1 on p. 184, the time span of an $\alpha(E)$ ensures that $\text{start}(\text{span}(\gamma_1)) = \text{end}(\text{span}(\gamma_1))$. Further, as a result of Lemma 8.2 on p. 184 assume that $\gamma_1$ represents the initiator occurrence of $\gamma_2$, that is, the following relation holds: $\text{end}(\text{span}(\gamma_1)) = \text{start}(\text{span}(\gamma_2))$; in other words, given, for example, a conjunction $E_a = E_1 \Delta E_2$ (where $\alpha(E_a) = E_1 \lor E_2$) $\gamma_1$ is assumed to represent the initiator occurrence and not the terminator occurrence. Then, there are two cases based on the part $\gamma_1 \prec \gamma_2 \land \neg (\gamma_1 \parallel \gamma_2)$ of the precedent of the ca-rule:

1. Successors to $\gamma_1$ of type $E_1$ (e.g., a terminator occurrence that could have been an initiator occurrence) do not render the precedent true, because $\forall \gamma \left( \text{type}(\gamma) = E_1 \land \gamma_1 \prec \gamma \Rightarrow \neg (\gamma \prec \gamma_2 \land \neg (\gamma \parallel \gamma_2)) \right)$ holds. Considering Defs 6.24 to 6.26 on p. 123, this means that either the relation $\text{end}(\text{span}(\gamma)) > \text{start}(\text{span}(\gamma_2))$ or $\text{(end}(\text{span}(\gamma)) = \text{start}(\text{span}(\gamma_2)) \land c_{\text{ord}}(\text{type}(\gamma)) > c_{\text{ord}}(\text{type}(\gamma_2))$ holds. In other words, for no event occurrence of $E_1$ succeeding the currently evaluated $\gamma_1$, for which the precedent is true, can the precedent of the ca-rule hold true.

2. Predecessors to $\gamma_1$ of type $E_1$ can render the precedent of the ca-rule true, because $\forall \gamma \left( \text{type}(\gamma) = E_1 \land (\gamma_1 \succ \gamma \lor \gamma_1 = \gamma) \Rightarrow \gamma \prec \gamma_2 \land \neg (\gamma \parallel \gamma_2) \right)$ holds. Considering Defs 6.24 to 6.26 on p. 123, this means that either the relation $\text{end}(\text{span}(\gamma)) < \text{start}(\text{span}(\gamma_2))$ or the relation $\text{(end}(\text{span}(\gamma)) = \text{start}(\text{span}(\gamma_2)) \land c_{\text{ord}}(\text{type}(\gamma)) < c_{\text{ord}}(\text{type}(\gamma_2))$
8.2 Proof of Early Detection of Expiration

holds. In other words, for any event occurrence of $E_1$ preceding $\gamma_1$ or occurring at the same time the precedent of the ca-rule may hold true.

Q.E.D.

Effectively, this lemma implies that event expressions such as conjunction, whose initiator transformation is a disjunction of the contributing event types, guarantees correct evaluation of $E_b$. For example, assume that $E_x = E_1 \Delta E_2$ and $E_a = E_x(\text{expire: } 10)$ is monitored and, at time 5, $G = \{\Gamma(E_1, [0,0]), \Gamma(E_2, [5,5])\}$. The initiator transformation is $\overline{\alpha}(E_x) = E_1 \triangledown E_2$. At time 5, there are two enabling intervals initiated (one at time 0 and one at time 5), but only one chronicle aperiodic event occurrence $E_b$ is generated for $\Gamma(E_x, [0,5])$. The reason is that only the first holds true for $\gamma_1 \prec \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2)$.

The second enabling interval starting at time 5 does not enclose $\gamma_2$.

**Lemma 8.5**

Any $E_b = CA(\overline{\alpha}(E_x), E_x, (\overline{\alpha}(E_x)+T)!)$ generates the same output as $E_a = E_x(\text{expire: } T)$ in any event context given that $c_{\text{ord}}(\overline{\alpha}(E_x)) < c_{\text{ord}}((E_x + T)!) < c_{\text{ord}}(E_x)$

**Proof:** Let there be an earliest available occurrence $\gamma_x \in G$ such that $\text{type}(\gamma_x) = E_x$. Using the ca-rule (in Def. 8.1 on p. 181), substitute $E$ with $E_b$, $E_1$ with $\overline{\alpha}(E_x)$, $E_2$ with $E_x$, $E_3$ with $(\overline{\alpha}(E_x)+T)!$, and let $\gamma = \Gamma(E_b, \text{span}(\gamma_2))$, then Eq. 8.1 is obtained:

\[
\forall E_b \forall \gamma_1, \gamma_2 \in G : \\
\text{type}(\gamma_1) = \overline{\alpha}(E_x) \land \text{ca ej}_\alpha(E_b, \gamma_1, \{\overline{\alpha}(E_x)\}) \land \text{type}(\gamma_2) = E_x \land X(E_b, \gamma_2, \{E_x\}, \{E_x\}) \\
\gamma_1 \prec \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2) \\
\neg \exists \gamma_3 \in G (\text{type}(\gamma_3) = (\overline{\alpha}(E_x)+T)! \land \gamma_3 \notin I^A(E_b) \land \\
\gamma_3 \succ \gamma_1 \land \neg(\gamma_3 \parallel \gamma_1) \land \gamma_3 \prec \gamma_2) \\
G \cup = \{\gamma\} \quad U_a(E_b) \cup = \{\langle \gamma_2, \gamma \rangle\} \quad U_a(E_b) \cup = \{\langle \gamma_2, \gamma \rangle\}
\]

Given Lemma 8.3 on p. 185, the context predicate only holds true for $\gamma_2 = \gamma_x$, since all previous event occurrences of $E_x$ have already been consumed regardless of event context. N.B. we do not substitute $\gamma_2$ with $\gamma_x$ for pedagogical reasons.
The predicate $\forall \gamma_1 \in \mathcal{G} (\text{start}(\text{span}(\gamma_1)) = \text{end}(\text{span}(\gamma_1)))$ holds true, since $\overline{\alpha}(E_x)$ is an instantaneous event according to Lemma 8.1 on p. 184. Let $\text{end}(\text{span}(\gamma_1)) = \text{start}(\text{span}(\gamma_x))$; in other words, let the initiator expression occurrence have the same termination time as the start of the composite event occurrence $\gamma_x$. The reason for this assumption is twofold. Firstly, for all $\gamma_1$ occurrences where $\text{end}(\text{span}(\gamma_1)) > \text{start}(\text{span}(\gamma_x))$ then either $\gamma_1 \prec \gamma_2$ or $\neg (\gamma_1 \parallel \gamma_2)$ is false according to Lemma 8.4 on p. 186.

Secondly, for all $\gamma_1$ occurrences where $\text{end}(\text{span}(\gamma_1)) < \text{start}(\text{span}(\gamma_x))$ then there are two cases (cf. $r$-rule in Def. B.4 on p. 367): (i) $\text{end}(\text{span}(\gamma_x)) \leq \text{end}(\text{span}(\gamma_1)) + T$, and (ii) $\text{end}(\text{span}(\gamma_x)) > \text{end}(\text{span}(\gamma_1)) + T$. Case (i) is due to that the enabling interval initiated by a previous occurrence of $\overline{\alpha}(E_x)$ has not terminated yet. It is of no consequence if an earlier initiated enabling interval is employed to compose an event occurrence of $E_b$ for two reasons: (a) any such enabling interval has a remaining duration that is less than $T$; and (b) a contributing $E_x$ occurrence is consumed when it is used to form an $E_b$ occurrence. Case (ii) means that once an $E_x$ occurrence has been used in one interval, it cannot be reused in another interval to form an $E_b$ occurrence. Case (ii), does not cause a problem, since that interval has expired and cannot be chosen according to $ca-e_i_{\alpha}$.

Given this choice of $\gamma_1$ in the $ca$-rule, we can substitute the $\gamma_1 \prec \gamma_2 \land \neg (\gamma_1 \parallel \gamma_2)$ with $true$, since it is true for $\gamma_2 = \gamma_x$. Further, $true$ can be removed. Applying this substitution and removal to Eq. 8.1 on the previous page results in Eq. 8.2.

$$\forall E_b \forall \gamma_1, \gamma_2 \in \mathcal{G} :$$

$$\text{type}(\gamma_1) = \overline{\alpha}(E_x) \land \text{ca-ei}_{\alpha}(E_b, \gamma_1, \{\overline{\alpha}(E_x)\}) \land \text{type}(\gamma_2) = E_x$$

$$\begin{array}{l}
X(E_b, \gamma_2, \gamma_2, \{E_x\}, \{E_x\}) \\
\neg \exists \gamma_3 \in \mathcal{G} \ (\text{type}(\gamma_3) = \overline{\alpha}(E_x) + T \land \gamma_3 \notin I^A(E_b) \land \gamma_1 \prec \gamma_3 \land \neg (\gamma_1 \parallel \gamma_3) \land \gamma_2 \succ \gamma_3) \\
\mathcal{G} = \{\gamma\} \quad \mathcal{U}_\alpha(E_b) = \{\langle \gamma_2, \gamma \rangle\} \quad \mathcal{U}_\omega(E_b) = \{\langle \gamma_2, \gamma \rangle\}
\end{array}$$

(8.2)

Considering the $r$-rule (in Def. B.4 on p. 367), there can only be a $\gamma_3 \in \mathcal{G}$ where $\text{type}(\gamma_3) = \overline{\alpha}(E_x) + T$ if there is an $\gamma_1 \in \mathcal{G}$ where $\text{type}(\gamma_1) = \overline{\alpha}(E_x)$ and $now \geq \text{end}(\text{span}(\gamma_1)) + T$. Since $\text{start}(\text{span}(\gamma_x)) = \text{end}(\text{span}(\gamma_1))$
8.2 Proof of Early Detection of Expiration

and start(span(γ2)) = start(span(γ3)), then there are two cases: (i) now < start(span(γx)) + T, and (ii) now ≥ start(span(γx)) + T.

Given case (i), then ¬∃γ3 ∈ G (type(γ3) = \overline{α}(E_x) + T ∧ γ3 \not∈ I_α(E) ∧ γ1 < γ2 ∧ ¬(γ1 || γ3) ∧ γ2 > γ1) is true, since there is no such event occurrence in G. Since end(span(γx)) ≤ now, then in the worst case end(span(γx)) = now. By using this worst-case situation, we can substitute now with the function end(span(γx)) so that now < start(span(γx)) + T is the same as end(span(γx)) < start(span(γx)) + T. This latter equation can be transformed to end(span(γx)) − start(span(γx)) < T. By using Def. 4.13 on p. 52, we obtain |span(γx)| < T. In other words, the duration of an occurrence of Eb is less than T.

Given case (ii), then ¬∃γ3 ∈ G (type(γ3) = \overline{α}(E_x) + T ∧ γ3 \not∈ I_α(E) ∧ γ1 < γ3 ∧ ¬(γ1 || γ3) ∧ γ2 > γ3) is false. By using similar transformations as in case (i), this predicate is false when |span(γx)| ≥ T. In this case, the condition in the hypothesis of the ca-rule is false.

Therefore, ¬∃γ3 ∈ G (type(γ3) = \overline{α}(E_x) + T ∧ γ3 \not∈ I_α(E) ∧ γ1 < γ3 ∧ ¬(γ1 || γ3) ∧ γ2 > γ3) can be replaced with |span(γx)| < T, since γ2 = γx. By substituting ¬∃γ3 ∈ G (type(γ3) = \overline{α}(E_x) + T ∧ γ3 \not∈ I_α(E) ∧ γ1 < γ3 ∧ ¬(γ1 || γ3) ∧ γ2 > γ3) with |span(γ2)| < T Eq. 8.3 is obtained from Eq. 8.2 on the facing page.

∀Eb ∀ γ1, γ2 ∈ G :

\begin{align*}
\text{type}(γ1) &= \overline{α}(E_x) ∧ \text{ca}(Eb, γ1, \{ \overline{α}(E_x) \}) ∧ \text{type}(γ2) = E_x \\
X(E_b, γ2, γ2, \{ E_x \}, \{ E_x \}) \\
|\text{span}(γ2)| &< T
\end{align*}

(8.3)

G∪={γ} \quad U_α(E_b)∪={γ2, γ} \quad U_α(E_b)∪={γ2, γ}

Since γ1 is not used anymore, it can be removed resulting in Eq. 8.4 (obtained from Eq. 8.3).

∀Eb ∀ γ2 ∈ G :

\begin{align*}
\text{type}(γ2) &= E_x \\
X(E_b, γ2, γ2, \{ E_x \}, \{ E_x \}) \\
|\text{span}(γ2)| &< T
\end{align*}

(8.4)

G∪={γ} \quad U_α(E_b)∪={γ2, γ} \quad U_α(E_b)∪={γ2, γ}
To summarize, Lemma 8.5 on p. 187 holds, since all event occurrences generated by $E_b$ have the same interval as $E_x$ event occurrences that must be less than $T$. 

N.B. all initiator occurrences of $E_x$ can be safely removed from time $\text{start}(\text{span}(\gamma_x)) + T$, since it cannot form another composite event occurrence of $E_b$.

**Theorem 8.1**

Given that all event types have an expiration time and that each primitive event type has a minimum interarrival time, then event composition is bounded.

**Proof:** Since Lemma 8.5 on p. 187 guarantees that $E_b$ generates the same output as $E_a$, then the timeout closing the interval of the chronicle aperiodic operator can be used to prune the filtered event log represented by connector sets. Since each event type is associated with an expiration time, then each connector set is bounded. Since the filtered event log is realized as connector sets, then it is also bounded. Since the filtered event log is bounded, then event composition is also bounded. 

Q.E.D.

### 8.3 Expiration Management

There are two methods for managing expiration:

1. each event type with an expiration time sets the timeout and when a timeout occurs the event types prune its associated connector sets and signal timeout data to all contributing event types

2. all contributing event types inherit the expiration time of composite event types and each event type sets timeouts independently of each other

The second method is the least complex method, since it reduces the number of dependencies between event types in the implementation. To define the inheritance of the second method, Def. 8.3 on the facing page that defines the transitive closure of (primitive) event type is necessary.
8.3 Expiration Management

Definition 8.3
Transitive closure:

\[
\text{trans} \_\text{closure}(E) = \begin{cases} 
\{E\} & \text{iff } \text{prim}(E) \\
\left( \bigcup_{E' \in \text{all operands}(E)} \text{trans} \_\text{closure}(E') \right) \cup \text{all operands}(E) & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

In other words, the \text{trans} \_\text{closure}(E) are all contributing event types of an event type.

The Def. 8.4 defines the relation of the expiration of the \text{trans} \_\text{closure}(E)
and \(E\) itself.

Definition 8.4
Expiration Inheritance:

\[
\forall E' \in \text{trans} \_\text{closure}(E) (\text{expire}(E') = \text{expire}(E))
\]

In other words, all contributing event types must have equal expiration time as the event type they contribute to.

Theorem 8.2
Any event type whose expiration times fulfill Def. 8.4 results in correctly composed event occurrences.

Proof: There are two cases:

(i) \(\exists E' \in \text{trans} \_\text{closure}(E) (\text{expire}(E) > \text{expire}(E'))\)

(ii) \(\exists E' \in \text{trans} \_\text{closure}(E) (\text{expire}(E) < \text{expire}(E'))\).

In case (i), consider \(E = (E_1 \triangle (\text{expire}:2)E_2);(\text{expire}:5)E_3\) in chronicle event context and \(G = \{\Gamma(E_1, [1,1]), \Gamma(E_1, [3,3]), \Gamma(E_2, [4,4]), \Gamma(E_3, [5,5])\}\), then
\[ \Gamma(E_1 \triangle (\text{expire}: 2)E_2, [3,4]) \text{ and } \Gamma(E, [3,5]) \text{ are generated since } \Gamma(E_1, [1,1]) \text{ is removed because } \text{end}(\text{span}(\Gamma(E_2, [4,4]))) - \text{start}(\text{span}(\Gamma(E_1, [1,1]))) > \text{expire}(E_1 \triangle (\text{expire}: 2)E_2). \] This may be incorrect, since \( \Gamma(E, [1,5]) \) may be the correct result since \( \text{expire}(E) = 5 \). A similar example can be provided for case (ii). If and only if Def. 8.4 on the previous page holds, is it guaranteed that event composition can produce correct results in all event contexts.

**8.4 Results**

In this chapter, the exact semantics of early detection of expiration of initiators is defined for any event expression in any event context. Strictly, each event type must be associated to a relative temporal event expression that is initiated when the monitored event expression is initiated and is terminated when the expiration time has occurred. When this relative temporal event occurrence is signalled, the monitored event type should prune the appropriate connector sets.

The initiating event expression of the relative temporal event expression is a formally expressed transformation of the monitored event expression. More specifically, it is a disjunction of primitive event types that can initiate the monitored event expression. For some event expressions in some contexts, this means that superfluous timeouts are initiated. For example, monitoring \( E = E_1 \triangle (\text{expire}: 10, \text{context}: \text{chronicle})E_2 \) implies that for each event occurrence of \( E \) there are two timeouts. Since chronicle event context is consuming contributing event occurrences, then one of the timeouts is unused.

However, in an event context that reuses contributing event occurrences, the timeouts are not superfluous. For example, consider the event expression \( E = E_1 \triangle (\text{expire}: 10, \text{context}: \text{recent})E_2 \) and assume that \( \mathcal{G} = \{ \Gamma(E_1, [0,0]), \Gamma(E_2, [5,5]), \Gamma(E_1, [15,15]) \} \); then \( \mathcal{G}' = \mathcal{G} \cup \{ \Gamma(E, [0,5]) \} \), but the \( \Gamma(E, [5,15]) \not\in \mathcal{G}' \) since the duration of \( \Gamma(E, [5,15]) \) is equal to 10. In this case, the second timeout initiated by \( \Gamma(E_2, [5,5]) \) is not superfluous.

**8.4.1 Discussion**

The suggested expiration management leads to that a common subexpression may well be of two different event types with the same base event type. For example, given that \( E = E_1; E_2 \) and \( E_a = E; (\text{expire}: 3)E_3 \) and \( E_b =
8.5 Related Work on Bounded Resources

\[ E \triangle (\text{expire: } 5) E_4, \] then this is equivalent to \( E' = E_1; (\text{expire: } 3) E_2, E'' = E_1; (\text{expire: } 5) E_2, E_a = E'; (\text{expire: } 3) E_3, \] and \( E_b = E'' \triangle (\text{expire: } 5) E_4. \)

One interesting issue is that it is possible to represent non-occurrences of initiated composite event types (representing omission failures). This is possible, since the expiration time is an attribute of each type. Concerning non-occurrences of initiated composite event occurrences, it can be useful to separate between expiration time (\( T_{\text{expire}} \)) and failure time (\( T_{\text{failure}} \)) where the \( T_{\text{failure}} \leq T_{\text{expire}} \) for each event type. For example, assume that a road is monitored where \( E_1 \) and \( E_2 \) are two sensors along the road. Moreover, assume that these sensors can read the registration number of the monitored cars. Assume that we are monitoring the event expression \( E = E_1; (\text{expire: } T_{\text{expire}}, \text{context: } \text{chronicle}, \text{failure: } T_{\text{failure}}) E_2 \) and that \( G = \{ \Gamma(E_1, [t, t]) \} \). We may obtain a failure indication that does not represent an actual omission, but represents a response time failure. For example, if the car passes by the \( E_2 \) sensor at a time larger than \( t + T_{\text{failure}} \) but less than \( t + T_{\text{expire}} \) this indicates that the car succeeded to pass by \( E_2 \) but not within the expected duration. Therefore, it can be interesting to first get a notification that the car has not passed within the expected time to alert subscribers that there may be an omission, but still leave the possibility that the car may pass the second sensor before the expiration time (representing a response time failure).

8.5 Related Work on Bounded Resources

In Chronicle recognition [DGG93, Dou96], a Chronicle deadline is used to abort the composition if it has not been terminated. Dousson et al. have provided no formal proof of that it actually works, but has a convincing argument that it works. Also, the argument only holds in general event context (both overlapping and non-overlapping). In contrast, this chapter has a formal proof that guarantees that expiration works for any event context.

In the monitoring of real-time logic expressions, Mok et al. [ML97a, ML97b] proved that their algorithm is resource-predictable. In contrast to the proof in this chapter, their work is only concerned about the recent and chronicle event contexts. Therefore, our proof is more general than their proof.

Concerning other event specification languages, only Carlson et al. [CL03] are concerned about resource-predictability, an issue that they only briefly looked into and will address in future work.
In contrast to our own work, in the licentiate thesis [Mel98a] the expiration was provided by an additional operator named within such that \( E \ within \ T \) meant the same as \( E \) where \( \text{expire}(E) = T \). There are two problems with the specification: it can be interpreted both as an event operator or as an attribute of an event operator (the latter was chosen [Mel98a]). If it is interpreted as an event operator, then it is difficult to specify how to handle expiration, since each event operator requires different treatment; as discussed in Ch. 6 expiration is modelled in the context predicates instead. If it is interpreted as an event attribute, then it is difficult to understand to which event type the expiration specification belongs.

8.6 Summary

To summarize, this chapter has demonstrated that event composition of any event expression in any event context in Solicitor has bounded resource requirements if each composite event type is associated with an expiration time and each primitive event type is sporadic. This proof is performed by symbolical manipulation of the generation rule of the chronicle aperiodic event operator. The chronicle aperiodic event operator is a necessary refinement of the aperiodic event operator, since the latter operator cannot handle overlapping enabling intervals. An important issue briefly addressed is the combination of common subexpressions and expiration time; basically, we advocate that common subexpressions are different event types if they inherit different expiration times from the event types they contribute to.
Algorithmic time complexity is defined as the asymptotic efficiency of algorithms according to, for example, Cormen, Leiserson, Rivest, and Stein [CLRS01]. In this chapter, the asymptotic efficiency of algorithms implementing event composition in different event contexts is investigated.

This study of algorithmic time complexity emphasizes event composition. That is, the time it takes to generate a number of composite event occurrences. This emphasis is significant, since each composite event occurrence requires selection of contributing event occurrences, composition of a (new) composite event occurrence as well as pruning of (contributing) event occurrences. Pruning could be studied in isolation; however, pruning is dependent on the number of generated composite event occurrences. Therefore, it is conjectured that, in the worst case, the algorithmic time complexity of pruning is in the same class as composition.

In this chapter, the least constrained evaluation of operator rules is assumed (recapitulated in the following text, originally found in Def. 6.11 on p. 109), since the impact of scheduling evaluation of event types has insignificant impact on the algorithmic time complexity of event composition. For example, difference between the control-driven scheduling of evaluation proposed in Section 7.2 on p. 167 and the least constrained evaluation is
that in the former ordering of evaluation is arbitrary and in the latter it is ordered among interdependent event types. The overhead of this static ordering is considered insignificant with respect to evaluation of operator rules (i.e., event composition).

Recapitulated definition:

**Definition 6.11**

**Least constrained event composition:** This algorithm is a refinement of the algorithm presented by Nilsson [Nil82, p. 21] depicted in Fig. 6.2 on p. 101. Let \( s \) be the state of event composition; let the \( \text{condition}(s, r, E) \) predicate be true if the condition of rule \( r \) evaluated on event type \( E \) is true in \( s \) (precise definition in Def:s 6.15 to 6.16 on p. 113); let \( \text{conclusion}(s, r, E) \) return a state change (precise definition in Def:s 6.15 to 6.16 on p. 113), let \( I \lambda = (\text{later\_state}(s, s_0) \lor s = s_0) \), and let \( C \lambda = r \in \text{rules}(E) \land \text{condition}(s, r, E) \) then

\[
\{{\{I}\}}
\text{while } \exists E, r (C) \text{ do begin}
\{{\{I \land \exists E, r (C)\}}
\text{select } E, r \text{ such that } C
\{{\{I \land C\}} \quad /\ast C \Rightarrow \exists E, r (C) \ast/
\text{s := conclusion}(s, r, E); /\ast \text{ updates state } s \ast/
\{{\{I \land \text{later\_state}(s, s')\}} \quad /\ast \exists E, r (C) \text{ is undefined, } s' \text{ is earlier state } \ast/
\text{end}
\{{\{I \land \neg \exists E, r (C)\}}\}
\]

\[\Box\]

The main method used to define the complexity classes is to define this iteration in terms of linear recurrence relations for each event context; these relations are used to derive functions (by using characteristic polynomials as

\footnote{The \( C \) (a lexical substitution) is used in two different ways in this definition. Firstly, it is used in the while statement where \( E \) and \( r \) are properly bound. Secondly, it is used in the predicates between statements where \( E \) and \( r \) are bound to the variables declared in the algorithm.}
9.1 Assumptions about Processing

The rest of this chapter is organized as follows, first a simplified model of event composition computation is presented. From this model, the sources of complexity are identified and computational cost functions are defined in terms of the complexity sources. Some of these computational cost functions are defined in terms of recurrence relations defining the number of computational steps in each iteration. These recurrence relations are solved and the result, a closed function, is the computational cost function representing the recurrence relation. From these computational cost functions, the time complexity classes are derived for different significant situations and significant design choices. The significant situation that is addressed is whether there is a single primitive terminator occurrence or multiple primitive terminator occurrences that are evaluated. The former is associated with implicit invocation and the latter with explicit invocation of event composition (in Section 4.9.2 on p. 77). The addressed design choice is whether safe pointers (cf. Ch. 11) are employed to manage event occurrence parameters or not.

9.1 Assumptions about Processing

\[
\{I\} \\
\text{foreach } E \text{ in } E_m \text{ do begin} \\
\quad \text{foreach } r \text{ in } \text{rules}(E) \text{ do begin} \\
\quad \quad \text{while } (C) \text{ do begin} \\
\quad \quad \quad \{\{I \land C\}\} \\
\quad \quad \quad s := \text{conclusion}(s, r, E); \\
\quad \quad \quad \{\{I \land \text{later\_state}(s, s')\}\} \\
\quad \quad \end{block} \\
\quad \end{block} \\
\end{block} \\
\{\{I \land \lnot \exists E, r(C)\}\} \\
\]

Figure 9.1: Simplified realization of the least constrained event composition algorithm
To investigate the time complexity, a simple yet accurate model of the execution of event composition is needed. Let $E_m$ be the set of monitored event types and assume that “foreach $E$” iterates over the event types according to the control-driven scheduling in Def. 7.13 on p. 173. Further, assume that we only consider usage (consumption) and not invalidation, since invalidation is dependent on generation that in turn is dependent on the usage of event occurrences. Given these assumptions, then a simplified realization of Def. 6.11 on p. 109 is shown in Fig. 9.1 on the previous page. In other words, for each rule of each event type event occurrences are generated as long as the condition $C$ is true.

Additional constraints on the evaluation of operator rules has to be introduced to make the model more accurate with respect to an implementation of event composition. For example, it is not feasible to use monotonically growing sets such as $G$ in an implementation, since there is no limit on the amount of memory they require.

It is assumed that each event operator combines an initiator and a terminator occurrence of distinct event types, since Solicitor does not have any event operators that combine more than two event occurrences at a time. Further, these event operators subsume event operators that do not combine event occurrences of distinct event types such as disjunction (in Def. 6.46 on p. 139) and use each event occurrence of the same contributing event type as both initiator and terminator at the same time.

First of all, the least constrained algorithm needs to be expanded to support the complexity study. The condition $(s, r, E)$ part (in Def. 6.11 on p. 109) can be replaced with $\text{range}(s, r, E)$ (in Def. 6.15 on p. 113 and Def. 6.16 on p. 113) effectively ignoring the hypothesis of the operator rule $(H)$, since it is not necessary to study the particular constraint added by an event operator in the complexity study. The reason is that given all possible patterns (i.e., permutations of temporal ordering) of (repeated) primitive event occurrences, there is some pattern that coerces the algorithm into its worst-case behavior regardless of the constraint introduced by the event operator.

The predicate $\text{range}(s, r, E)$ can be replaced with its definition for generation rules, since only composition is studied. Therefore, in this chapter $\text{range}(s, r, E)$ is replaced with $\exists \gamma_1, \ldots, \gamma_n \in G (R \land R_c)$. This predicate can be replaced with $\exists \gamma_1, \gamma_2 \in G (R \land R_c)$, since only an initiator and terminator of distinct event types are considered. Moreover, the conclusion $(s, r, E)$ can be replaced with $G \cup \{\gamma\}, U_{\alpha}(E) \cup = \{\langle \gamma_1, \gamma \rangle\}, U_{\omega}(E) \cup = \{\langle \gamma_2, \gamma \rangle\}$. 
9.1 Assumptions about Processing

It is assumed that an implementation is based on the connector sets introduced in Section 7.1 on p. 152. Therefore, the $\exists \gamma_1, \gamma_2 \in \mathcal{G}(R \land R_c)$ can be replaced with $\exists \gamma_1 \in \mathcal{Q}_{E_1}(E), \exists \gamma_2 \in \mathcal{Q}_{E_2}(E)(R_c)$ since $\forall E_i (\mathcal{Q}_{E_i}(E) \subseteq \mathcal{G} \land \forall \gamma \in \mathcal{Q}_{E_i}(E)(\text{type}(\gamma = E_i))$ and $R$ is only used to ensure the types of candidate event occurrences ($\gamma_i$).

If we replace $\mathcal{G} \cup \{\gamma\}$ with an iteration assigning the connector sets that the event $E$ is contributing to, then the algorithm can be rewritten as in Fig. 9.2 (where $\Delta$ denotes a program segment).

```
{\{I\}}
foreach $E$ in $\mathcal{E}_m$ do begin
  foreach $r$ in rules($E$) do begin
    while ($\exists \gamma_1 \in \mathcal{Q}_{E_1}(E), \gamma_2 \in \mathcal{Q}_{E_2}(E) (R_c)$) do begin
      $\{\{I \land \hat{C}\}\}$
      $\mathcal{U}_a(E) \cup = \{(\gamma_1, \gamma)\}, \mathcal{U}_o(E) \cup = \{(\gamma_2, \gamma)\}$;
      $\gamma = \Gamma(E, [\text{start}(\text{span}(\gamma_1)), \text{end}(\text{span}(\gamma_2))] \}$
      $\Delta_P$
      foreach $E_{\text{parent}}$ in $\{E' | E \in \text{all operands}(E')\}$ do begin
        $\Delta_C \{ \mathcal{Q}_E(E_{\text{parent}}) \cup = \{\gamma\} \}$;
        end
    $\{\{I \land \text{later state}(s, s')\}\}$
  end
end
```

Figure 9.2: Algorithm performing composition only

This specialized algorithm illuminates what happens in composition. The segment $\Delta_E$ is iterated once for each event type. The segment $\Delta_r$ is iterated once for each rule associated with the (principal operator of the) event type. Since only composition is considered, there is a constant number of iterations over rules per event type, because there is a constant number of eligible generation rules per operator. The segment $\Delta_G$ is iterated once for each pair of eligible initiator and terminator event occurrences. The connector sets are assumed to be implemented as queues, where event occurrences are inserted in the order they will be used. Therefore, the cost of locating an initiator and terminator event occurrence is assumed to be
constant \((O(1))\). The segment \(\Delta_C\) is iterated over once for each event type \(E\) is contributing to. Since common subexpressions are problematic, they are assumed not to be employed; for example, a common subexpression used in two different composite event type specifications with different expiration times implies that this common subexpression represents two different event types (cf. Section 8.4.1 on p. 192). Therefore, it is assumed that \(\Delta_C\) is only iterated over once. Finally, the segment \(\Delta_P\) addresses the generation of a new event occurrence.

Since the selection of event occurrences for matching is based on the context predicate, only these predicates need to be studied for algorithmic time complexity. The reason is that these predicates provide the following information:

1. the number of initiator event occurrences there are in a connector set

2. the number of initiator event occurrences that can be matched with a single terminator event occurrence

With these two facts, it is conjectured that the algorithmic time complexity of any correct implementation that fulfills the invariants (in Ch. 7) can be derived.

### 9.1.1 Implementation Assumptions

It is assumed that the queues realizing the connector sets contain pointers to event occurrences representing the event occurrences. Internally, it is assumed that an event occurrence is a directed acyclic graph of event occurrences, where the edges are pointers from one event occurrence to another. These pointers can either be unsafe or safe. The safe pointers resolve one of the major problems of the unsafe pointers; the major problem that is solved is management of dangling pointers.²

Moreover, it is assumed that the delivery is delayed until all event composition resulting from a set of terminators has been completed.
9.2 Sources of Complexity

The notation used in this chapter is as follows: a hat on a variable (e.g., $\hat{e}$) means that it is an integer variable that denotes the size of some input, an upper case variable within ‘|’ denotes a size of something (e.g., $|E|$ denotes the size of an event type), and a tilde on a variable means that it is a constant (e.g., $\tilde{c}$). The sources of complexity in event composition are defined in Table 9.1. N.B. that the size of the filtered event log is three variables that are defined by the following relationship; let $c_1, c_2 \in \mathbb{R}$ such that $0 \leq c_1 + c_2 \leq 1$, then $i = c_1 \hat{g}$ and $t = c_2 \hat{g}$. That is, the filtered event log is assumed only to consist of initiators and terminators. When $i$ and $t$ are subscripted, the subscript denotes the step in the event composition processing.

9.2.1 Motivation of Complexity Sources and their Model Coverage

The more event occurrences there are in the filtered event log, the more steps may have to be taken to generate composite event occurrences ($\hat{g}$).

---

Table 9.1: Variables for Time Complexity Investigation

<table>
<thead>
<tr>
<th>Name</th>
<th>Variable</th>
<th>Constant</th>
<th>Segment</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of filtered event log</td>
<td>$\hat{g} (i, t)$</td>
<td>$</td>
<td>\hat{g}</td>
<td>$</td>
</tr>
<tr>
<td>Event type size</td>
<td>$\hat{e}$</td>
<td>$</td>
<td>E</td>
<td>$</td>
</tr>
<tr>
<td>Number of monitored event types</td>
<td>$\hat{m}$</td>
<td>$</td>
<td>\mathcal{E}_m</td>
<td>$</td>
</tr>
<tr>
<td>Size of parameters</td>
<td>$\hat{p}$</td>
<td>$</td>
<td>P</td>
<td>$</td>
</tr>
</tbody>
</table>
For example, in the general event context a terminator event occurrence causes a composite event occurrence to be generated for each initiator event occurrence in the filtered event log. Also, it is necessary to separate initiators from terminators in the algorithmic time complexity argument until the end.

Moreover, the more contributing event types there are, the more steps may have to be taken to complete event composition processing ($\hat{e}$). For example, assume that $E_1; (E_2; E_3)$ is monitored and when $now = 2$ the event history $G = \{\Gamma(E_1, [0,0]), \Gamma(E_2, [1,1]), \Gamma(E_3, [2,2])\}$ is explicitly evaluated; this situation leads to that both an $\Gamma(E_2; E_3, [1,2])$ occurrence is generated as well as an $\Gamma(E_1; E_2; E_3, [0,2])$ occurrence. Two composite event occurrences are generated regardless of the event context in two iterations of the least constrained evaluation algorithm.

The third source of complexity is the number of monitored event types ($\hat{m}$). The more event types that are monitored, the more iterations may have to be performed by the evaluation algorithm. N.B., the second and third source are two different aspects of related issues. The relation is that all contributing event types of an event expressions are members of the $E_m$. However, it is necessary to distinguish them, because $\hat{m}$ represents all event types in the system, whereas $\hat{e}$ only represents the currently evaluated event type.

The fourth source of complexity is the (physical) size of parameters for each primitive event type ($\hat{p}$). The more data that is carried, the longer time it takes to process this data. For example, when a composite event occurrence is signalled to a recipient, the parameters must be copied into a message containing the composite event occurrence for transmission to the recipient.

To summarize, there are four important sources that are addressed within this investigation of algorithmic time complexity. The application requirements determine which source is important.

Concerning model coverage there are two segments that have not been addressed: $\Delta_r$ and $\Delta_C$. The number of operator rules associated with event operators are fixed and, moreover, we only consider composition (that addresses a subset of operators rules for each event type). Thus, a parameter representing the $\Delta_r$ is considered to be uninteresting. The $\Delta_C$ is not covered since common subexpressions are, as discussed earlier, unlikely and, thus, they do not have a great impact on the upper bound time complexity.
9.3 Time Complexity Basics

Lemma 9.1
In the complexity investigation, \( \hat{i} \) and \( \hat{t} \) can be substituted with \( \hat{g} \).

**Proof:** When \( \hat{g} \to \infty \), then \( \hat{i} \to \infty \land \hat{t} \to \infty \) since they both are assumed to be fractions of \( \hat{g} \) (in Section 9.2 on p. 201). Since complexity analysis is based infinitesimal calculus, this substitution can be used. \( \text{Q.E.D.} \)

9.3 Time Complexity Basics

This study considers the upper bound complexity (in Def. 9.1) as defined by, for example, Cormen et al. [CLRS01, p. 44]. Their definition is:

**Definition 9.1**
Asymptotic upper bound:
\[
O(g(n)) = \{ f(n) | \forall n \in \mathbb{N} \exists c, n_0 \in \mathbb{N} (n \geq n_0 \Rightarrow 0 \leq f(n) \leq cg(n)) \}
\]

Tight bounds (in Def. 9.2) and lower bounds (in Def. 9.3) are not considered, since it is hard to make meaningful claims. For example, the lower bound for event composition is that no event occurrences result from it. In the best case, one iteration per terminator is needed regardless of all other parameters.

**Definition 9.2**
Asymptotic tight bound:
\[
\Theta(g(n)) = \{ f(n) | \forall n \in \mathbb{N} \exists c_1, c_2, n_0 \in \mathbb{N} (n \geq n_0 \Rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)) \}
\]

**Definition 9.3**
Asymptotic lower bound:
\[
\Omega(g(n)) = \{ f(n) | \forall n \in \mathbb{N} \exists c, n_0 \in \mathbb{N} (n \geq n_0 \Rightarrow 0 \leq cg(n) \leq f(n)) \}
\]
Their abuse of equality is adopted as well where, for example, \( f(n) = O(g(n)) \) means \( f(n) \in O(g(n)) \), since it has advantages in discussing complexity. For example, \( n^2 + O(n) = O(n^2) \) means that no matter what the anonymous function chosen in \( O(n) \) is, it is always possible to choose a function in \( O(n^2) \) such that the equality holds.

### 9.3.1 Semantics of Notation of Multiple Parameters

Since there is no single most significant parameter in event composition that is universally accepted, the complexity definition usually includes more than one parameter combined with arithmetic operators. The interpretation is that given that a parameter is the most significant parameter in an application, then the asymptotic efficiency is dependent on that parameter as if the other parameters are constants.

For example, \( O(\hat{e}^2\hat{p}) \) expression means that if \( \hat{e} \) is the most significant parameter, then the algorithm scales polynomially with respect to the size of an event expression. In contrast, if \( \hat{p} \) is the most significant parameter, then the algorithm scales linearly with respect to the parameter size. In both cases, the other parameter is considered to be constant.

### 9.4 Algorithmic Time Complexity of Event Monitoring

The total cost function for event monitoring can be modeled as Def. 9.4. The lookup is performed before the composition defined in Fig. 9.2 on p. 199 and \( \text{prepare
deliver} \) is performed after the composition. The cost function composition is derived from Fig. 9.2 on p. 199.

**Definition 9.4**

**Monitoring cost function**:

\[
\text{monitor}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \text{lookup}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) + \\
\text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) + \\
\text{prepare
deliver}(\hat{g}, \hat{e}, \hat{p}, \hat{m})
\]

\[\square\]
The cost function lookup is the computation cost of finding the connector set each primitive event occurrence should be added to, the cost function composition is the computational cost of performing event composition and the cost function prepare delivery is the computational cost for preparing event occurrences for delivery to subscribers.

**Definition 9.5**

**Lookup cost function:**

\[
\text{lookup}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \begin{cases} 
O(\hat{g} \log(\hat{m})) & \text{iff multiple terminators} \\
O(\log(\hat{m})) & \text{iff single terminator}
\end{cases}
\]

Def. 9.5 is assumed to be based on an efficient structure such as a balanced tree or a hash structure optimized for reading, in which the lookup of the representation of the primitive event type associated with a primitive event occurrence scales with \(\log(\hat{m})\). If there are multiple occurrences that are evaluated, then this is multiplied by \(\hat{g}\).

### 9.4.1 Composition Cost Function

As mentioned in the introduction to this chapter, the number of steps in event composition can be defined as a set of recurrence relations. These recurrence relations can be used to derive the complexity class from the closed-form function\(^3\) that are derived from the recurrence relation. This closed-form function is used to derive the complexity class.

In our (simplified) model of event composition, the \(\hat{e}\) denotes the \(\hat{e}\)th iteration of \(\Delta_E\). To simplify the interpretation of the reasoning, consider the following example. Let \(E_m = \{E_i|1 \leq i \leq |E| \land E_i = E_{\alpha(i-1)}; E_{i-1}\} \cup \{E_{\alpha i}|1 \leq i \leq |E|\}\) (where \(E_{\alpha i}\) is the \(i\)th initiator event type), for example, \(E_3 = E_{\alpha 2}; E_2, E_2 = E_{\alpha 1}; E_1\). When an \(\Gamma(E_1, [t,t'])\) is signalled, this can result in an \(\Gamma(E_2, [t_2,t'])\) occurrence (in step \(\hat{e} = 1\)). This can, in turn, lead to an \(\Gamma(E_3, [t_3,t'])\) occurrence (in step \(\hat{e} = 2\)) etc. That is, in this simplified

\(^3\)A closed-form function does not rely on the previous values in a series, only on the \(n\) where \(n\) is the order of the value in the series.
model of event composition the $\hat{e}$th processing step is the evaluation of event type $E_{\hat{e}}$; N.B., in reality there is no such relation.

**Definition 9.6**

**Composition cost function:** Let $s_{\hat{e}}$ be the number of processing steps for the $\hat{e}$th event type in an event expression in the cost function composition($g, \hat{e}, \hat{p}, \hat{m}$) = $s_{\hat{e}}$ where $s_{\hat{e}}$ = context($\hat{t}_{\hat{e}-1}, \hat{t}_{\hat{e}-1}$) generate($\hat{e}$) + $s_{\hat{e}-1}$ defines the number of computational steps that have to be performed for evaluating an expression of $\hat{e}$ event operators.

\[ \square \]

The context: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ defines how many times $\Delta_r$ and $\Delta_g$ are iterated. The generate: $\mathbb{N} \rightarrow \mathbb{N}$ is the number of steps taken to generate a composite event occurrence in $\Delta_P$.

**Simplification of the Recurrence Relation**

The recurrence relation in Def. 9.6 contains too many unknown variables to be solved. However, since this is a study of algorithmic time complexity where asymptotic efficiency is considered, we can introduce simplifications that are valid in the calculus used in asymptotic efficiency.

First of all, $\hat{t}_{\hat{e}-1}$ can be substituted with $\hat{t}$ assuming that $\forall \hat{e}(\hat{t}_{\hat{e}} = \hat{c}_1 \hat{t})$ where $\hat{c}_1 = \hat{c}_1 / |E|$, because when $\hat{t} \rightarrow \infty$, then $\forall \hat{e}(\hat{t}_{\hat{e}} \rightarrow \infty)$. In other words, when the size of the filtered event log approaches $\infty$, then the total number of initiators as well as number of initiators per contributing event type approaches $\infty$.

Moreover, for recent, chronicle and continuous event context, assume that $\forall \hat{e}(\hat{t}_{\hat{e}} \leq \hat{t})$. In other words, there are no number of terminator occurrences of an event type that are more than the number of initiator occurrences for this event type. This implies that $\hat{c}_1 > \hat{c}_2$, where $\hat{c}_2 = \hat{c}_2 / |E|$ implying that $\hat{c}_1 > \hat{c}_2$.

Given these simplifications, the recurrence relation (in Def. 9.6) can be written as

\[ s_{\hat{e}} = \text{context}(\hat{t}, \hat{t}_{\hat{e}-1}) \text{ generate}(\hat{e}) + s_{\hat{e}-1} \quad (9.1) \]

The $\hat{t}_{\hat{e}}$ variable is substituted for each addressed event context in the following sections.
9.4 Algorithmic Time Complexity of Event Monitoring

Generation of Event Occurrences

There are two cases: (i) unsafe pointers, and (ii) safe pointers. To avoid dangling pointers for unsafe pointers, it is necessary to traverse the whole directed acyclic graph representing a composite event occurrence and copy it. For safe pointers, this is not necessary. The generation cost function is defined as:

Definition 9.7
Generate cost function:

\[
generate(n) = \begin{cases} 
(\hat{p} + 1)n + \hat{p} & \text{iff unsafe pointers are used} \\
1 & \text{iff safe pointers are used}
\end{cases}
\]

For the unsafe pointers, the recurrence relation defining generate(n) is

\[x_n - x_{n-1} = \hat{p} + 1\]

implying that for each contributing event type/processing step we need to copy the parameters of the contributing event occurrence into the composite event occurrence being constructed. The closed-form function of \(n\) for this recurrence relation where \(x_0 = 1\) is \((\hat{p} + 1)n + \hat{p}\).

This is the series: \(\hat{p}, 2\hat{p} + 1, 3\hat{p} + 2, 4\hat{p} + 3, \ldots\), that is, for one operator the cost of generation is \(2\hat{p} + 1\) (two primitive event types plus construction of one composite event occurrence), for two operators the cost of generation is \(3\hat{p} + 2\) (three, one for each contributing primitive event type, plus one for the occurrences representing composite event occurrences) etc. The proof of the equivalence between the recurrence relation and its closed-form function is left as an exercise for the reader.

For safe pointers, the cost of generation is always '1' since it is not necessary to make a deep-copy of the complete event occurrence until the event occurrences are to be signalled across an address boundary. A deep-copy is performed in prepare delivery before signalling for both safe and unsafe pointers.

Number of Terminator Occurrences

A composite event occurrence from one processing step is assumed to be a terminator in the next processing step. Different event contexts give different results.
Definition 9.8

Context cost function:

\[
\text{context}(\hat{i}, \hat{t}_{\hat{e} - 1}) = \begin{cases} 
\hat{t}_0 & \text{iff } \hat{e} = 1 \\
\text{composed}(\hat{i}, \hat{t}_{\hat{e} - 1}) & \text{iff } \hat{e} > 1 
\end{cases}
\]

□

In the first iteration, at most \(\hat{t}_0\) unprocessed terminator occurrences are processed. In all the following processing steps, the number of processing steps depends on the number of generated terminator occurrences from the last processing step.

Definition 9.9

Number of generated event occurrences function:

\[
\text{composed}(\hat{i}, \hat{t}_{\hat{e} - 1}) = \begin{cases} 
\hat{t}_{\hat{e} - 1} & \text{for chronicle event context} \\
1 & \text{for recent event context} \\
\hat{i} & \text{for continuous event context} \\
\hat{t}_{\hat{e} - 1} & \text{for general event context}
\end{cases}
\]

□

The explanation of \textit{composed} function is in the following four paragraphs.

**Chronicle:** Def. 6.39 on p. 129 is true only for unconsumed, non-expired event occurrences. The number of generated event occurrences is defined by \(\text{composed}(\hat{i}, \hat{t}_{\hat{e} - 1}) = \min(\hat{i}, \hat{t}_{\hat{e} - 1})\), because no more than the minimum if either initiator or terminator occurrences can be used to form composite event occurrences. This can be reduced to \(\text{composed}(\hat{i}, \hat{t}_{\hat{e} - 1}) = \hat{t}_{\hat{e} - 1}\), since \(\forall \hat{e} (\hat{t}_{\hat{e}} \leq \hat{i})\) is assumed as discussed in Section 9.4.1 on p. 205.

**Recent:** Since \(\forall \hat{e} (\hat{t}_{\hat{e}} \leq \hat{i})\) is assumed, then there is either one or zero event occurrences generated that is used by the next processing step:

\[
\text{composed}(\hat{i}, \hat{t}_{\hat{e} - 1}) = \begin{cases} 
1 & \text{iff } \hat{t}_{\hat{e} - 1} > 0 \\
0 & \text{iff } \hat{t}_{\hat{e} - 1} = 0
\end{cases}
\]
9.4 Algorithmic Time Complexity of Event Monitoring

Since \( t_0 > 0 \), then \( \text{composed}(i, \hat{t}_{\hat{e}-1}) = 1 \).

**Continuous:** For continuous event context, \( \text{composed}(i, \hat{t}_{\hat{e}-1}) = i \) since a terminator occurrence can be reused to construct \( \hat{t}_e \) composite event occurrences, whereas terminator occurrences are consumed.

**General:** Since each combination of initiator and terminator occurrence can form a composite event occurrence, then \( \text{composed}(i, \hat{t}_{\hat{e}-1}) = \hat{t}_{\hat{e}-1} \).

### 9.4.2 Prepare for Delivery Cost Function

The prepare_delivery\( (\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \text{generate}(\hat{e}) \) for multiple terminator case for both unsafe and safe pointers, since it assumed that a composed event occurrence must be delivered outside the address boundary. The composite event occurrence must be serialized into a structure that can be passed to another address space (on some physical node in a distributed system). Then, Def. 9.10 presents this cost function.

**Definition 9.10**

\[
\text{prepare\_delivery}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) =
\begin{cases}
\hat{e}(\hat{p} + 1) + \hat{p} = O(\hat{e}\hat{p}) & \text{iff single terminator} \\
\hat{g}(\hat{e}(\hat{p} + 1) + \hat{p}) = O(\hat{g}\hat{e}\hat{p}) & \text{iff multiple terminators}
\end{cases}
\]

\[\square\]

### 9.4.3 Algorithmic Time Complexity for Event Contexts

The general simplified recurrence relation in Eq. 9.1 on p. 206 can be rewritten as in Eq. 9.2 by replacing \( \text{generate}(\hat{e}) \) with its definition (in Def. 9.7 on p. 207).

\[
s_{\hat{e}} - s_{\hat{e}-1} =
\begin{cases}
\text{context}(i, \hat{t}_{\hat{e}-1})(\hat{e} + \hat{p}) & \text{iff unsafe pointers} \\
\text{context}(i, \hat{t}_{\hat{e}-1}) & \text{iff safe pointers}
\end{cases}
\]

In chronicle, recent, and continuous event context, the closed-form functions can be derived from these non-homogeneous difference equations by substi-
tuting \( e \) with \( n \) and \( s \) with the function \( q(n) \) where \( q(n) = n^k \sum_{i=0}^{n} c_i \cdot n^i \) is a trial function. For unsafe pointers, the result of this substitution is Eq. 9.3.

\[
q(n) - q(n - 1) = \text{context}(i, i_{n-1})((\hat{p} + 1)n + \hat{p}) \tag{9.3}
\]

By setting \( q(n) = n(c_0 + c_1n) = c_0n + c_1n^2 \) for unsafe pointers, Eq. 9.4 can be obtained.

\[
n(c_0 + c_1n) - (n - 1)(c_0 + c_1(n - 1)) = \text{context}(i, i_{n-1})((\hat{p} + 1)n + \hat{p}) \tag{9.4}
\]

By treating \( c' = \text{context}(i, i_{n-1}) \) as a constant with respect to \( n \) for the moment, Eq. 9.5 is the result. After the composition function is defined, we address the cost function of event contexts.

\[
n(c_0 + c_1n) - (n - 1)(c_0 + c_1(n - 1)) = c'(\hat{p} + 1)n + \hat{p} \tag{9.5}
\]

Eq. 9.5 can be transformed into Eq. 9.6 by arithmetic operations.

\[
2c_1n + c_0 - c_1 = c'(\hat{p} + 1)n + c'\hat{p} \tag{9.6}
\]

Finally, Eq. 9.6 can be transformed into Eq. 9.7.

\[
c_0 - c_1 - c'\hat{p} = (c'(\hat{p} + 1) - 2c_1)n \tag{9.7}
\]

A closed-form function can be found for all \( n \) by setting \( c_i \) for \( n = 1 \) so that the equality holds [GS93]. This leads to Eq. 9.8 which defines these constants.

\[
c_1 = c'(\hat{p} + 1)/2 \\
c_0 = c'/2(\hat{p} + 1) + c'\hat{p} = c'/2(3\hat{p} + 1) \tag{9.8}
\]

The closed function is:

\[
q(n) = \frac{c'}{2}(3\hat{p} + 1)n + \frac{c'}{2}(\hat{p} + 1)n^2 = \frac{c'}{2}((3\hat{p} + 1)n + (\hat{p} + 1)n^2) \tag{9.9}
\]

Since composition\((\hat{g}, \hat{e}, \hat{p}, \hat{m}) = s_{\hat{e}} = q(\hat{e})\), according to Def. 9.6 on p. 206 and the fact that \( s_n = q(n) \), then

\[
\text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \frac{c'}{2}((3\hat{p} + 1)\hat{e} + (\hat{p} + 1)\hat{e}^2) \tag{9.10}
\]

To order the variables as in the function parameter list, then Eq. 9.10 can be rewritten as Eq. 9.11 on the facing page.
9.4 Algorithmic Time Complexity of Event Monitoring

\[ \text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \frac{c'}{2}(\hat{e}(3\hat{p} + 1) + \hat{e}^2(\hat{p} + 1)) \]  

(9.11)

For safe pointers, similar transformations can be used for the trial function \( q(n) = c_0 + c_1n \). Again, if the context function is treated as constant \( c' \), then the closed-form function is \( q(n) = c' * n + c_0 \), where \( q(0) = c_0 \). In this case, \( q(0) = 0 \) giving \( q(n) = c'n \). This yields the composition function:

\[ \text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = c'\hat{e} \]  

(9.12)

**Time Complexity of Event Composition in Chronicle Event Context**

<table>
<thead>
<tr>
<th># Terminators</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unsafe</td>
</tr>
<tr>
<td>Single</td>
<td>( O(\log(\hat{m}) + \hat{e}^2\hat{p}) )</td>
</tr>
<tr>
<td>Multiple</td>
<td>( O(\hat{g}(\log(\hat{m}) + \hat{e}^2\hat{p})) )</td>
</tr>
</tbody>
</table>

Table 9.2: Time complexity of event composition in chronicle event context

Table 9.2 defines the complexity for unsafe and safe pointers for processing a single terminator as well as processing multiple terminators.

**Unsafe pointers:** The complexity class of unsafe pointers is derived as follows. Let \( c' = \hat{t}_{\hat{e}-1} \) (from Def. 9.9 on p. 208) in Eq. 9.8 on the facing page, then Eq. 9.13 is the result.

\[
\begin{align*}
c_1 &= (\hat{p} + 1)\hat{t}_{\hat{e}-1}/2 \\
c_0 &= (3\hat{p} + 1)\hat{t}_{\hat{e}-1}/2
\end{align*}
\]  

(9.13)

Then, by using Eq. 9.11 \( \text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \hat{t}_{\hat{e}-1}/2(\hat{e}(3\hat{p} + 1) + \hat{e}^2(\hat{p} + 1)) \). Due to Lemma 9.1 on p. 203 \( \hat{t}_{\hat{e}-1} \) can be substituted with \( \hat{g} \). Therefore, for multiple terminators, Eq. 9.14 can be derived from Eq. 9.11.

\[ \text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \hat{g}/2(\hat{e}(3\hat{p} + 1) + \hat{e}^2(\hat{p} + 1)) = O(\hat{g}\hat{e}^2\hat{p}) \]  

(9.14)
For a single occurrence of the terminator, then \( \hat{t}_{\hat{e}-1} = \hat{t}_0 = 1 \) that gives Eq. 9.15 derived from Eq. 9.11 on the previous page.

\[
\text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \frac{1}{2}(\hat{e}(3\hat{p} + 1) + \hat{e}^2(\hat{p} + 1)) = O(\hat{e}^2\hat{p}) \quad (9.15)
\]

The total cost function for chronicle context for multiple terminators is derived by replacing lookup (in Def. 9.5 on p. 205), composition (in Eq. 9.14 on the previous page, and prepare_delivery (in Def. 9.10 on p. 209) cost functions in monitor (Def. 9.4 on p. 204) giving Eq. 9.16 for processing multiple terminator occurrences.

\[
\text{monitor}(\hat{g}, \hat{e}, \hat{p}, \hat{m})) = O(\log(\hat{m})) + O(\hat{g}\hat{e}^2\hat{p}) + O(\hat{g}\hat{e}\hat{p}) = O(\hat{g}(\log(\hat{m}) + \hat{e}\hat{p})) \quad (9.16)
\]

For processing a single terminator occurrence, Eq. 9.17 is derived.

\[
\text{monitor}(\hat{g}, \hat{e}, \hat{p}, \hat{m})) = O(\log(\hat{m})) + O(\hat{e}^2\hat{p}) + O(\hat{e}\hat{p}) = O(\log(\hat{m}) + \hat{e}^2\hat{p}) \quad (9.17)
\]

**Safe pointers:** In this case, \( \text{composition}(\hat{g}, \hat{e}, \hat{p}, \hat{m}) = \hat{e}_{\hat{e}-1}\hat{e} \). Due to Lemma 9.1 on p. 203, \( \hat{e}_{\hat{e}-1} \) is substituted with \( \hat{g} \). For processing multiple terminator occurrences, Eq. 9.18 defines the total cost.

\[
\text{monitor}(\hat{g}, \hat{e}, \hat{p}, \hat{m})) = O(\log(\hat{m})) + O(\hat{g}\hat{e}) + O(\hat{g}\hat{e}\hat{p}) = O(\hat{g}(\log(\hat{m}) + \hat{e}\hat{p})) \quad (9.18)
\]

Similarly to the unsafe pointer case, the complexity class for evaluation of a single terminator can be derived from the case addressing evaluation of multiple terminators yielding:

\[
\text{monitor}(\hat{g}, \hat{e}, \hat{p}, \hat{m})) = O(\log(\hat{m}) + \hat{e}\hat{p}) \quad (9.19)
\]
9.4 Algorithmic Time Complexity of Event Monitoring

9.4.4 Recent Event Context

The complexity of recent context is the same as for chronicle event context. The reason is that any optimizations for recent event context does not give any effect on the complexity class.

9.4.5 Continuous Event Context

<table>
<thead>
<tr>
<th># Terminators</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>$O(\log(m) + \hat{g}^2\hat{p})$</td>
</tr>
<tr>
<td>Multiple</td>
<td>$O(\hat{g}(\log(m) + \hat{e}^2\hat{p}))$</td>
</tr>
</tbody>
</table>

Table 9.3: Complexity of continuous event context

Table 9.3 defines the complexity for unsafe and safe pointers for the processing a single terminator as well as processing multiple terminators (of the same event type).

The reasoning is the same as for to chronicle event context with the difference that context($\hat{g}, \hat{e}, \hat{p}, \hat{m}$) = $\hat{i}$ rather than context($\hat{g}, \hat{e}, \hat{p}, \hat{m}$) = $\hat{t}_{\hat{e}-1}$. This makes a difference for processing of a single terminator, since the time complexity class of event composition in continuous event context is always dependent on the size of the filtered event log. Due to Lemma 9.1 on p. 203, $\hat{i}$ is substituted with $\hat{g}$.

9.4.6 General Event Context

For general event context, the scalability is of $O(\log(x) + \hat{g}^6\hat{p})$ for both unsafe and safe pointers. The safe pointers give better constants in the cost function, but the functions scale in the same way.

Let context($a, b$) = $a^b$, then $s_{\hat{e}} - s_{\hat{e}-1} = ((\hat{p}+1)\hat{e} + \hat{p})$ context($\hat{i}, \hat{t}_{\hat{e}-1}$) can be transformed into $s_{\hat{e}} - s_{\hat{e}-1} = ((\hat{p}+1)\hat{e} + \hat{p})\hat{i}\hat{t}$. By expanding this equation, $s_{\hat{e}} - s_{\hat{e}-1} = (\hat{p} + 1)\hat{e}\hat{t} + \hat{p}\hat{e}\hat{t}$ can be obtained. By only studying the $\hat{p}\hat{e}\hat{t}$, it is possible to obtain the closed-form function that is of less complexity than the actual closed-form function. Therefore, assume that $s_{\hat{e}} - s_{\hat{e}-1} = \hat{p}\hat{e}\hat{t}$ is considered instead. The solution to this is $q_n = n^2(\hat{i}-1)/(\hat{i}-1)$. That is, this partial solution is of complexity $O(\hat{g}^6)$. Since the actual closed-form function scales worse than this, it is also of exponential complexity.
9.5 Discussion

This discussion contains the relevance of this study and remaining issues.

9.5.1 Relevance of Study

This study of complexity is important, since it provides guidelines of how event contexts affect the processing. Out of the sources of complexity (in Section 9.2 on p. 201), the size of the filtered event log (\(\hat{g}\)) is likely to be the most significant, since systems with event types with short interarrival times and expiration times that are many factors longer than the interarrival times are likely. For example, in a phone switch system, event occurrences may arrive with an interarrival time of 1ms, but the (soft) expiration time for meeting any of these events may be counted in seconds. Hence, in the worst case it may be necessary to store a few thousands of event occurrences for on-line analysis. The response time for both recent and chronicle event contexts are of \(O(1)\) with respect to the \(\hat{g}\) for processing a single terminator occurrence.

In contrast, \(\hat{e}\) is not considered to be significant except in general event context, since most event expressions contain at most 2-5 operators. The reason is that it is difficult to understand event composition as is evident in, for example, the work by Berndtsson et al. [BMH99]. The problem is that in addition to the problem of understanding predicates, there are event contexts that entail that it is necessary to understand the history of processing. Without proper tools for generating, verifying, debugging, and understanding event types, it is unlikely that there will be larger event expressions.

Moreover, the \(\hat{p}\) is likely to be insignificant for safe pointers since there will be at most as many parameters as there is in an ordinary function or procedure call plus system parameters such as process identifier, transaction identifier, task identifier. Also, instead of passing complete copies of objects, it is possible to only pass the reference to the object. Due to these reasons, it is conjectured that this will not affect the processing significantly.

Although \(\hat{p}\) can be insignificant, safe pointers is an important design concept that should be considered in the design of an event monitor, since it reduces the complexity class from polynomial to linear complexity. However, this impact is dependent on \(\hat{e}\) rather than \(\hat{p}\).
9.5.2 Remaining Issues

There are a number of issues not considered in this study. Space complexity of the amount of memory needed to maintain event composition has not been addressed in this complexity study. However, it is conjectured that it scales in a similar fashion to the explicit invocation of event composition (the multiple terminator case in this investigation). The non-overlapping event contexts are not covered, however, it is conjectured that these belong to the same algorithmic time complexity class as the overlapping counterpart. The reason for this conjecture is that non-overlapping is only a filter that does not change the processing of the selection of candidate event occurrences. Finally, a more esoteric issue is to solve the recurrence relation for general event context. It is not necessary, since solving a part of the recurrence relation results in the exponential complexity class with respect to $\hat{e}$ and polynomial with respect to $\hat{g}$. Therefore, solving the whole recurrence relation will only result in that it is of exponential complexity.

Introducing non-order of event occurrences in the connector sets increases the algorithmic time complexity. This has not been addressed in this model. However, any relaxation of the order must consider how this affects the algorithmic time complexity. For example, the continuous event context may be significantly affected by non-order, since it scales linearly with respect to $\hat{g}$ for a single terminator occurrence.

9.6 Related Work on Algorithmic Complexity

The only work addressing algorithmic time complexity of the presented event specification languages are Chronicle recognition and monitoring of real-time logic expressions. Space complexity has been addressed in, for example, REACH [Deu94].

9.6.1 Chronicle Recognition

Dousson et al. [DGG93], Dousson [Dou96] and Ghallab [Gha96] claim that the time complexity of Chronicle recognition is in practice (average case) $O(Km^2)$ where $m$ is the number of event types in a Chronicle specification (equivalent to $\hat{e}$) and $K$ is the number of evaluated situations (partially composed event occurrences), where the $K$ is proportional to $\hat{g}$. Therefore, the average complexity class of is $O(\hat{g}\hat{e}^2)$. 
However, in the worst case, the complexity is $O(\hat{g}^\hat{e})$. Consider the following example:

**chronicle $E$**

\[
E_1 < E_2 < E_3 < E_4
\]

end

If there are $n_1$ $E_1$ occurrences, then there will be $n_1$ instantiated copies of the graph representing $E_c$. Then, if these event occurrences are succeeded by $n_2$ $E_2$ occurrences, then there will be $n_1n_2$ instantiated copies. Further, if these are succeeded by $n_3$ $E_3$ occurrences, then there will be $n_1n_2n_3$ instantiated copies. Let $\hat{g} = n_1 = n_2 = n_3$, then this example shows that the number of copies grows exponentially with respect to $\hat{e}$ in the worst case. In other words, Chronicle recognition is in the same complexity class as general event context in the worst case.

The importance of their complexity result is that given appropriate time constraints and a tractable environment, it is possible to have a scalability behavior that is useful for real-time systems. This is supported by experiments on an actual application [DGG93]

### 9.6.2 Monitoring of Real-time Logic

The major concern in the monitoring of real-time logic is to reduce the complexity of the matching part of event processing. That is, given an event occurrence, then does it match any current evaluation of an expression in real-time logic? In their work, they have reduced this from $O(n^3)$ to $O(n)$ in the supported event contexts (recent and chronicle) (for a class of real-time logic expressions avoiding the $\#$ function [LMK98]), where $n$ is the number of nodes in the graph representing real-time logic expressions. Each event part (e.g., $\!(E, i)\!$) is a node and each constraint (e.g., $\!(E_1, i) < \!(E_2, i)\!$) is a vertex. For example, in $n = 2$ in the expression $\!(E_1, i) < \!(E_2, i) \lor \!(E_1, i) + T < \!(E_2, i)\!$. This is equivalent to $E_1;\langle context : chronicle, expire : T\rangle E_2$. The $n$ is conjectured to be proportional to $\hat{e}$, since each event operator in Snoop can be translated into a non-variable-sized real-time logic expression according to Liu et al. [LMK98]. Thus, the complexity class of their matching is in the worst case $O(\hat{e}^3)$ and in the best case $O(\hat{e})$.

The composition of the equivalent Solicitor expression is of $O(\log(\hat{m}) + \hat{e} \bar{p})$, where $\log(\hat{m}) + \bar{p}$ is to be considered a constant in this comparison. Therefore, Solicitor scales as good as or better than real-time logic with respect to the size of an event type in the chronicle event context.
Moreover, other factors, in particular $\hat{g}$, have not been addressed in their study. However, $\hat{g}$ does not matter in recent and chronicle event contexts for processing a single terminator occurrence.

## 9.7 Summary

To summarize, recent and chronicle event contexts scales, at worst, linearly with respect to $\hat{e}$ and $\hat{p}$ for safe pointers for processing a single terminator occurrence. Continuous event context, scales, at worst, linearly with respect to $\hat{g}$, $\hat{p}$, and $\hat{e}$ for safe pointers. Finally, general event context scales polynomially with respect to $\hat{g}$ and exponentially with respect to $\hat{e}$.

For unsafe pointers in recent, chronicle, and continuous event contexts, the algorithmic time complexity is polynomial with respect to $\hat{e}$.

The processing of multiple terminator occurrences scales linearly with respect to $\hat{g}$ for event composition in recent, chronicle, and continuous event contexts.
Part III

Architectural Issues in Event Monitoring
This part is theme 2 and addresses efficient designs for monitoring event occurrences. Ch. 10 introduces an overall software architecture for event monitoring in embedded real-time systems that meets goals G21 and G22. Ch. 11 introduce a novel architecture for memory management for event composition addressing goal G24.
“There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies.”

/C.A.R. Hoare

Design is always a trade-off between different quality attributes such as performance, timeliness, dependability etc. In this chapter, an architecture for predictable event monitoring for embedded systems is presented. The guiding principles of this architecture is timeliness and performance.

10.1 Overview of the Architecture

The architecture is depicted in Figs 10.1 to 10.3 on pp. 224–225. It follows the client-server architecture style [GS93], where the event monitoring clients are either application tasks executing instrumented code that signals event occurrences to event monitoring (the service) or (application) tasks subscribing to event types receiving event occurrences. The event monitoring service is provided by a non-empty set of monitor server tasks performing event composition scheduled by the conductor task (whose responsibility is inter-task scheduling of monitoring tasks). The partitioning of tasks to processors and memory regions is application dependent.
The application programming interfaces for clients (in Fig. 10.2), for implementing instrumented tasks and subscriber tasks, consist of two layers: a general layer and a layer derived from an event monitor specification. The responsibility of the general layer is to wrap the details of asynchronous message passing in procedures, simplifying the application programmer’s task. The responsibility of the specification-derived layer is to provide an interface closely related to the event monitor specification. The reason for the specification-derived layer is that the general layer is abstract with respect to event types to enable polymorphic management of any event type; further, the general layer contains low-level aspects with respect to the programming language for performance reasons. An example of the abstract aspect of the general layer is that it must provide accessibility to parameters of event expressions such as $E_1;E_2$ or $N(E_1,E_2;E_3,E_4)$, as well as provide signalling functionality for primitive event types that have different user parameters. In contrast, the specification-derived layer provides a (C++) class for each event type, enabling access to the event parameters. An example of the low-level aspects of the general interface is that it takes the physical size
10.1 Overview of the Architecture

Figure 10.3: Server layers of the event monitoring architecture

of event parameters as arguments. In contrast, the specification-derived interface provides a method in each class representing primitive event types for signalling event occurrences.

The monitoring tasks use the (five) server layers (in Fig. 10.3): (i) the message management, (ii) the event monitor management, (iii) the event type management (S3 on page 76), (iv) the event composition scheduling layer (S2 on page 76), and (v) a split layer consisting of event memory management, connector set management, subscription management, and timeout management. The message management (i) receives messages from clients via the general interface, decodes the message type and invokes the appropriate method in the event monitor layer. The event monitor layer (ii) consists of a singleton object (per task) that implements the actual treatment of the messages from the client such as “register primitive event type” or “signal primitive event type”. The event monitoring management invokes the event composition scheduling management to decide which event type to evaluate if the monitor is configured for control-driven scheduling (S2 on page 76). The responsibility of (iii) the event type management layer is to maintain event types as well as provide processing of event occurrences according to specified event operators in specified event contexts. Finally, the responsibility of the parts in layer (iv) are as follows: the event memory management is responsible for allocation and deallocation of memory for event instances (representations of the event occurrences) as well as aggregating event instances into messages for message passing (S5 on page 76); the connector set management is responsible for maintaining the connector sets between
event types (S4 on page 76); and subscription management is responsible for maintaining sets of subscribers, invoking construction of messages containing composite event occurrences (S4 on page 76); finally, the timeout service management is responsible for setting, generating, and canceling timeouts related to event types.

The client layers and the server layers are based on DeeDS operating system interface that provides task management, asynchronous message passing with explicit buffering, and memory management. This interface is an adapter to make applications source-portable across different target platforms that include OSE Delta, Linux, and Solaris.

10.1.1 Main Scenarios

There are two main scenarios from the perspective of the clients: (i) instrumented code signals a primitive event occurrence and (ii) a subscriber (task) receives a set of composite event occurrences packed in one (or more) messages.

Scenario (i): In the specification-derived interface, there is a class representing each event type, where each class representing a primitive event type has a method called “signal”. This method takes an argument that is an automatically generated structure (or record), derived from the associated event type, containing all event parameters of the primitive event type. The instrumented code assigns a variable of the parameter structure and uses this variable when the signal method is called. This method invokes the signal procedure in the general interface that, in turn, sends a message to a server. The server processes the incoming primitive event occurrence according to the event monitor specification. If there is a composite event occurrence that a subscriber (task) is interested in, then this composite event occurrence is signalled to the subscriber.

Scenario (ii): The subscriber uses a function call provided by the server-derived interface that blocks until event occurrences arrive. This function returns a set of event occurrences, the result from the last invocation of event monitoring.

10.1.2 Main Configurations

Depending on the configuration of the event monitor, scheduling of event composition (i.e., scheduling of evaluation of event types) is either data-driven or control-driven. In control-driven scheduling, signalling of the re-
resulting composite event occurrences from an event monitoring task is delayed until all composition in that task is completed. In data-driven scheduling, the default is to signal resulting event occurrences immediately to the subscriber, however, the delivery can be delayed in a similar fashion to control-driven scheduling. Finally, event composition can either be implicitly or explicitly invoked. In the former case, every time a primitive event occurrence is signalled event composition is invoked. In the latter case, event composition is invoked by ordering the event monitor to evaluate the set of unprocessed signalled primitive event occurrences.

10.1.3 Event Monitor Specification

The specification-derived interface is generated by a compiler. For example, the specification

\[ \text{event } e1(\text{int } p1). \]

results in the following C++ types:

```cpp
struct Event_e1_Parameters {
    int _p1;
};

class Event_e1_TypeInfo {
public:
    ...
    reflection service interface
    ...
}

class Event_e1 {
public:
    static const Event_e1_TypeInfo & dem_getTypeInfo();
    Event_e1_Parameters & get_Parameters();
    static void signal(Event_e1_Parameters&);
    ...
}
```

This source code is compiled together with the monitored object as well as the subscribers. The class provides interface to accessing event occurrences as well as signalling event occurrences.

The event monitor specification follows an extension of the grammar discussed in Syntax 4.1 on p. 43. First of all, a monitor consists of a set of monitoring tasks and primitive event specifications:
Definition 10.1

Event monitor specification:

\[
\text{Monitor} ::= \text{monitor identity [Attributes]} \{ \text{ EventGroupings } \}
\]

\[
\text{EventGroupings} ::= (\text{EMTask}|\text{PrimitiveEvents}) +
\]

Each event monitoring task groups the processing of a set of composite event types:

Definition 10.2

Event monitoring task:

\[
\text{EMTask} ::= \text{task identity [Attributes]} \{ \text{ EventSpecification } \}
\]

\[
\text{EventSpecification} ::= \text{event identity [Attributes]} [\text{ ParSpec } ]\text{ is } E \}
\]

Definition 10.3

Primitive event specification:

\[
\text{PrimitiveEvents} ::= \text{primitive } \{ \text{ PrimitiveEvent } \}
\]

\[
\text{PrimitiveEvent} ::= \text{event identity } [\text{ ParSpecs } ]\}
\]

Definition 10.4

Parameter specifications:

\[
\text{ParSpecs} ::= \text{ ParSpec } +, \text{ ParSpecs } | \epsilon
\]

\[
\text{ParSpec} ::= (\text{ int } | \text{ double } | \text{ char } ) \text{ identity } [\text{ integerConstant } ]
\]

A complete example of an event monitor specification for monitoring an elevator system with three floors is shown in Fig. 10.4 on the facing page. The primitive event types floor1 to floor3 (referred to as floorN) denote that somebody inserted a security card (with identity card), pressed a button and wishes the elevator to go to floor N. The primitive event type arriveN denotes that the elevator has arrived at floor N. Two examples of behaviors are presented. The behavior correct1 is signalled when there is a floor1 event occurrence followed by an arrive1 event occurrence without any intermediate emergency event occurrence. Any initiated composition of correct1 expires within 10 minutes. The incorrect1 behavior is that a
10.1 Overview of the Architecture

floor1 event occurrence is followed by a relative temporal event occurrence floor1+10min without any intermediate arrive1 event occurrence. Both of these specifications are monitored by event monitoring task taskOne whose default event context is chronicle.

```
monitor ElevatorMonitor {
    primitive {
        event floor1(int card).
        event floor2(int card).
        event floor3(int card).
        event arrive1.
        event arrive2.
        event arrive3.
        event emergency.
    }
    task taskOne(context: chronicle) {
        event correct1(int card) is
            N(expire: 10min)(floor1, emergency, arrive1).
        event incorrect1 is
            N(floor1, arrive1, floor1+10min).
        ... // repeat for each floor/arrive event type
    }
}
```

Figure 10.4: Event monitor specification example

Attributes defined on a higher level than the event expressions such as the language construct ‘monitor’ are inherited to the lower levels such as event monitor tasks and event types. What inheritance means depends on the attribute in question. For example, binding the context attribute to chronicle in the monitor implies that the default context is chronicle unless something else is stated. On the other hand, the expiration attribute is inherited according to Section 8.3 on p. 190.

A variant of this example based on parameterized event types (i.e., future work) is shown in Fig. 15.1 on p. 318 in Section 15.3.13 on p. 317. Basically, by using parameterization the event monitor specifications can be compacted.
10.2 Design Choices

This section presents the relationship between significant concepts such as tasks, transactions etc. as well as data structures and algorithms for predictable event monitoring. The design of memory management is in Ch. 11, since it is not based on existing techniques in the same manner as the rest of the services in the software architecture.

The solution starts with maintenance issues and continues with each of the services discussed in Section 4.9 on p. 75. This is followed by experiments indicating the effect of the proposed optimization, a discussion on achieving and improving the quality attributes timeliness and efficiency. Finally, we present the related work on architectural issues.

10.3 Maintenance Issues

There are two maintenance issues: (i) how to manage event monitoring in a development process, and (ii) how event composition can repair itself in the presence of failures (i.e., fault tolerance).

10.3.1 Management of Event Monitoring

There are three related issues in management of event monitoring. Firstly, adding and removing elements from a set of monitored event types, since this is the effect of software maintenance within event monitoring. Secondly, compilation of event types into an internal representation is desired, since it can significantly improve performance of event monitoring. Thirdly, instrumentation of the monitored object is a significant issue (cf. Section 2.2.1 on p. 20).

Maintenance can take place in two different situations: (i) while the monitored objects and event monitors are not in operation (off-line), or (ii) during operation of the system (monitored objects and event monitors). The former is assumed in this Ph.D. thesis, since it simplifies the solution of predictable event monitoring. The second situation is complicated without attempting to achieve timeliness, since it involves replacing existing tasks in a current state with tasks containing updated event types. For example, assume that $E_1;(\text{context: chronicle})E_2$ is replaced with $E_1;(\text{context: recent})E_2$, then what should be done with the connector set between $E_1$ and $E_1;(\text{context: chronicle})E_2$? Moreover, the formerly generated
10.3 Maintenance Issues

event occurrences have been produced in the chronicle context and now it should be switched to recent context? There are several issues concerning what is and is not allowed, how to implement it etc. Then, in a real-time system, this has to be performed while achieving timeliness. However, the design choices within the architecture cover the situation (ii) by allowing extensions. This second situation is considered future work and is further addressed in Section 15.3.8 on p. 315.

There is a compiler taking specification of event types into an internal representation. This compilation performs the following steps defined in Section 2.2 on p. 19, also addressed in Section 3.1.3 on p. 29:

- The compilation partially supports step 1a (sensor configuration), since a class for each primitive event type is generated. This class can be used to instrument the monitored object.
- All sensors are enabled in step 2.
- Analysis specification and analysis execution (step 4a and step 4c) for on-line usage are fully supported within the expressibility of Solicitor.

There is no display specification, since the purpose is not to display information for an operator, but to notify recipients of significant event occurrences. The instrumentation of the monitored object is performed by manually inserting software sensors in the code executed by the monitoring object.

10.3.2 Fault Tolerance

There are two issues of fault tolerance for event monitoring: (i) the management of patterns that may fail to occur due to failures, and (ii) fault tolerance of the event monitor itself. The first issue is handled with combinations of the expiration attribute of event types, the relative temporal event operator, the chronicle aperiodic and the aperiodic event operator, the non-occurrence operator, the logical occurrence operator, and the event contexts. Given these event operators, it is possible to specify timeouts relative to initiators, to specify enabling intervals where it is expected that an event occurrence should occur, to specify omissions with the non-occurrence operator, and, finally, it is possible to add guards. This provides a sufficient syntax in which we can express foreseeable behaviors as well as omissions failures in these foreseeable behaviors. For example, we can specify that a
remote procedure call request should be followed by a remote procedure call reply as well as that an omission of the reply is a failure.

The second issue is not considered in this Ph.D. thesis and is considered future work (in Section 15.3.9 on p. 316). For example, if a database is a monitored object and it crashes, then what should happen to the state of the event monitor during recovery? This is not a simple question. To our knowledge, only one article (by Hanson et al. [HCD+98]) addresses this issue for active databases and it does not provide a solution for recovery in active databases even when the event monitoring tasks are transactional tasks.

10.4 Issues Concerning Event Types, Tasks, and Transactions

Event types, tasks, and transactions are tightly coupled in event monitoring. First, the association of event types and tasks is addressed. This is followed by a discussion of the association of tasks and transactions, where management of the isolation property of transactions is a significant issue.

10.4.1 Association of Event Types and Event Monitoring Tasks

Event types can be grouped by event monitoring tasks (in Section 10.1.3 on p. 227). The composition processing of an event monitoring task of a signalled event occurrence is atomic. That is, once the composition is invoked in an event monitoring task, this task does not perform anything else before it has completed the processing of the signalled event occurrences. It is the responsibility of the conductor task to ensure that all critical event types are processed before new event occurrences are considered.

If there is a problem of schedulability of the task load, then it is assumed that either the time constraints can be renegotiated, that new levels of criticality can be introduced, or that the event types can be redefined.

10.4.2 Association of Tasks and Transactions

The monitored object can either be a resource (e.g., a processor (physical) or a data object in a database (logical)), or it can be a temporal scope (e.g., a task or a transaction). The event monitoring tasks may execute within transactions or outside transactions. Moreover, the recipients may be
executing within or outside transactions. If a recipient is executing within a transaction, then this can either be in the same transaction or a different transaction with respect to the monitored object.

10.4.3 Isolation Property of Transactions and Event Monitoring

In the proposed architecture, event monitoring tasks do not implicitly execute within transactions, since, for example, attempts to perform operations may be interesting to monitor for auditing purposes such as computer encroachment. However, the event monitoring tasks must meet the properties of transactions; in particular, the isolation property must be maintained. Therefore, the event occurrences carry sufficient data to maintain isolation, that is, at least the transaction identifier. N.B., there are several degrees of isolation suggested by Gray [GR94, section 7.6] to improve performance of tasks by relaxing the ACID properties (in Section 3.4.1 on p. 34). However, this is not covered in this Ph.D. thesis.

Given parameterized event types (future work in Section 15.3.13 on p. 317), we can transform any event type parameterized on transaction identifier into an event type that meets the isolation property. For example, assume that \( E \) is parameterized on the transaction identifier, then the expression \( E' = E; \text{transactionCommitCompleted} \) parameterized on the transaction identifier guarantees isolation if \( \text{transactionCommitCompleted} \) is signalled after a transaction has completed successfully.

10.5 Event Subscription

As mentioned in Section 4.9.3 on p. 78, subscription (between contributing and contributed event types as well as event types and external subscribers such as tasks or ECA rules) can be divided into inter-task subscription and intra-task subscription. In this Ph.D. thesis, it is assumed that intra-task subscription is realized as connector sets connecting contributing event types with their composite event types. As an optimization, event occurrences are not signalled to the event type in intra-task subscription, but annotated with a counter that is initialized to the number of subscribers that use this occurrence. Once a composite (subscribing) event type cannot use a contributing occurrence for another composite event occurrence, then this counter is decreased by one step. Once this counter reaches zero, then it is possible to
prune the connector set of that event occurrence. This optimization is currently only designed for a static set of event types that cannot change while the system is in operation.

In inter-task subscription, (asynchronous) message passing is used to signal the event occurrences to subscribers. If the sending and receiving tasks share the same address space, this can be optimized to sending only the reference rather than copying the message from one address range to another.

10.6 Event Pruning

Three issues are important for pruning: (i) management of concurrent event monitoring tasks. (ii) management of aggregation of timeouts, and (iii) management duplicates to improve performance of event monitoring.

10.6.1 Management of Concurrent Event Occurrences

As presented in Section 8.3 on p. 190, there are two ways of managing expiration. Either an expiration of a timeout is propagated to the contributing event types or the expiration time is inherited to the contributing event types. Given that event composition can be performed concurrently, then the latter is preferred, since there may be race conditions between the pruning functionality and signalling of event occurrences in the former solution. Inheriting also implies that an event type does not need to keep track of the initiated compositions of contributing event types. The additional computational cost is the management of timeouts for all event types have expiration time rather than only for the event types with explicit expiration time specifications.

Concurrent event monitors are a problem with respect to pruning, since pruning can affect several event monitoring tasks. For example, assume that we are monitoring $E_1; (\text{expire}: 10)E_2; E_3$ and that $E_2; E_3$ is monitored by task $x_1$ and $E_1; (\text{expire}: 10)E_2; E_3$ by $x_2$; moreover, assume now = 11 and $G = \{ \Gamma(E_1, [1,1]), \Gamma(E_2, [2,2]) \}$; at time 11 $\Gamma(E_3, [11,11])$ is added (to $G'$), then there is a race condition between the timeout and the event occurrence of $E_3$. Also, there may be a race condition between the pruning message sent from $x_1$ to $x_2$ and the $E_3$ occurrence.

Either these race conditions can be prevented, or detected and handled. An example of preventive methods is to order event types and to guarantee
that no processing is performed before all event occurrences are delivered. Another example is to schedule event monitoring tasks in such a way that no race conditions occur. An example of detection and recovery is to use death certificates. Once an event occurrence is invalidated it is marked with a death certificate. This death certificate is sent as the pruning message.

In this Ph.D. thesis, ordering of event types is employed together with guaranteed delivery of event occurrences with the same termination time. Detection and recovery of race conditions is considered future work (in Section 15.3.6 on p. 313).

10.6.2 Aggregation of Event Occurrences Based on Timeouts

In systems with a sparse time base, several initiated composite event occurrences may expire at the same time. In such systems, it may be useful to aggregate event occurrences and references to event occurrences (in the connector sets) so that it is possible to prune them with one operation rather than one operation per event occurrence.

10.6.3 Aggregation of Duplicates

In pruning, it is desirable to reduce the number of elements that has to be actually pruned by aggregating related event occurrences. For example, Dousson et al. [DGG93, Dou96] keep duplicates together so that only one operation is required to prune all related duplicates. This is possible for any event context that uses branching time (in Section 2.3 on p. 22), since there are event occurrences that are shared among composite event occurrences. In particular, if this is used together with a construct that can invalidate several event occurrences simultaneously (such as non-overlapping contexts or timeouts), then this is a useful optimization. This issue is consider future work since they are optimizations that are useful in specific situations (in Section 15.3.6 on p. 313).

10.7 Event Operators, Algorithms, and Mechanisms

Each kind of event operator in Solicitor is represented as a class where each event operator (of an event type) is an instance of this class. The realization
of the algorithm is done in the mechanism. In this Ph.D. thesis, the procedural mechanism is chosen since it can be realized from and verified with the proposed schema in this Ph.D. thesis (cf. Ch. 7).

10.8 Timeout Service

There are three main functions in the timeout service: (i) an event monitoring task can set a timeout for an event type, (ii) an event monitoring task can cancel a timeout for an event type, and (iii) a timeout can occur. The timeout service maintains a mapping of timeouts to a mapping of task to a set of event types. By using this data structure, it is possible to maintain aggregate information about timeouts. For example, if $E_1+T_1$ and $E_2+T_2$ is monitored by task $x$ then if $E_1+T_1$ sets a timeout at time $t_1$ and $E_2+T_2$ sets a timeout at time $t_2$ such that $t_1 + T_1 = t_2 + T_2$ then these two requests are aggregated into one timeout in the timeout service. When the timeout occurs, a single message containing the timeout information for both $E_1+T_1$ and $E_2+T_2$ are sent to $x$.

For simplicity, a centralized timeout service is assumed. In this case, race conditions, in terms of incoming timeout request, can be handled by letting the timeout server decide who wins races.

10.9 Event Parameter Management

Each event occurrence carries an event type identifier, the time interval in which it occurred, and user or application parameters (e.g., method parameters of the instrumentation of a method generating an event occurrence). To manage large objects, only object identifiers are passed as the parameter.

10.10 Optimizations

The scheduling and timeout service relies on sets of event types. To improve performance, these sets are realized as bitsets, where the index of the bit represents an event type. By using this method, a union of sets can be performed by doing a bitwise or, intersection of sets by doing a bitwise and etc. Moreover, a set of bits can be packed and sent in a message without linearization.
10.11 Discussion

There are two issues that are desirable to discuss. Firstly, the quality attributes timeliness, efficiency, and correctness. Secondly, the delay caused by splitting event monitoring into several tasks.

This architecture is designed with timeliness and efficiency in mind while achieving correct output. The important part for timeliness is to split event monitoring into one or more tasks, where each task monitors event types of the same criticality. This should provide a basis for schedulability. One issue has not been discussed previously: how to derive worst-case execution time of event monitoring tasks.

There is an upper bound on the computation since all event types must have an expiration time and the minimum interarrival time of primitive event occurrences is known. Since there is an upper bound, then worst-case execution times can be obtained. Since event composition is well-defined compared to the arbitrary computations addressed in worst-case execution time analysis (e.g., Pushner et al. [PS97]), then it is conjectured that a more accurate worst-case execution time can be obtained for event monitoring tasks than for arbitrary computations. However, this conjecture remains to be proven (in Section 15.3.7 on p. 314).

Concerning efficiency, several optimizations have been presented. In general, by aggregating data about event occurrences and avoiding sending this data until everything is processed speeds up the event composition. N.B. these optimizations affect the worst-case as well as the average-case execution times and, thus, all types of real-time systems benefit from them.

Concerning correctness, concurrency is an important issue since there may be race conditions. By letting each event monitoring task complete its event composition processing before delivering any results to other event monitoring tasks can correctness with respect to generating event occurrences be maintained. The ordering of event types ensure that timeouts are processed in the right order with respect to system requirements.

There is an inherent delay in intertask communication. This delay must be accounted for in the scheduling of event monitoring tasks. In general, it is desirable that the time granularity of the system is larger than the worst-case execution time of event composition.
10.12 Related Work

In this related work section, the following issues are compared and contrasted for each major approach:

- **Support for real-time systems**
  - expiration management
  - criticality management
  - mapping of event monitoring functionality to tasks

- **Support for distribution**
  - support for time and order
  - interoperability aspects
  - separation of global and local event monitoring

- **Support for transactions**

- **Event composition processing**
  - Concurrent event composition
  - Implicit or explicit invocation of event composition.
  - Data-driven or control-driven scheduling of event composition.
  - Immediate or delayed delivery of composite event occurrences.

- **Support for declarative monitoring**

The features of Solicitor are summarized in Table 10.1 on the facing page.

10.12.1 Snoop

There are several variations of Snoop (e.g., [CM94, CKAK94, Lia97, CYY98, Yan99, Ada02, AC02]). The first sources [CM94, CKAK94] address the semantics of Snoop and event parameter contexts in a centralized computer system with termination semantics, whereas later sources address distribution [Lia97, Yan99, YC99] as well as interval semantics [Ada02, AC02]. Concerning distribution, they address global and local event detection [Lia97] as well as timestamping [Yan99, YC99] (a refinement of Schwidersky’s work [Sch96]). The common base is the Snoop event specification language.
10.12 Related Work

<table>
<thead>
<tr>
<th>Issue</th>
<th>Solicitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for real-time systems</td>
<td></td>
</tr>
<tr>
<td>Expiration management</td>
<td>Yes</td>
</tr>
<tr>
<td>Criticality management</td>
<td>Yes</td>
</tr>
<tr>
<td>Mapping of event monitoring functionality to tasks</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for distribution</td>
<td></td>
</tr>
<tr>
<td>Time and order</td>
<td>Not addressed</td>
</tr>
<tr>
<td>Interoperability aspects</td>
<td>Not addressed</td>
</tr>
<tr>
<td>Separation of global and local event monitoring</td>
<td>Not addressed</td>
</tr>
<tr>
<td>Support for transactions</td>
<td>Not addressed</td>
</tr>
<tr>
<td>Event composition processing</td>
<td></td>
</tr>
<tr>
<td>Concurrent composition</td>
<td>Yes</td>
</tr>
<tr>
<td>Invocation</td>
<td>Configurable</td>
</tr>
<tr>
<td>Data-driven vs. control-driven scheduling</td>
<td>Configurable</td>
</tr>
<tr>
<td>Immediate or delayed delivery</td>
<td>Configurable</td>
</tr>
<tr>
<td>Pruning</td>
<td>Addressed</td>
</tr>
<tr>
<td>Support for declarative monitoring</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: Summary of Solicitor features

They do not address real-time issues in the sources. In terms of distribution, interoperability issues are not covered. Support for transactions are not discussed in these sources. Moreover, different invocation, scheduling, and delivery schemes are not addressed. At least one source, Liang [Lia97], addresses the support for declarative monitoring in terms of that there is a compiler in the system.

10.12.2 HiPAC, REACH, SAMOS, ODE, NAOS

HiPAC [DBB88, CBB89], REACH [BBKZ94, Buc94, Deu94, BZBW95], SAMOS [GD92, GD93, Gat94], ODE [GJM93, GJS93], and NAOS [CC96, Ron97] are all centralized active database prototypes, where transactions are an inherent part of rule processing encompassing event composition. Real-time features for event composition are not addressed in any of these sources, except that REACH [BZBW95] does not allow composite event types for real-time purposes and optimizes primitive event detection by bypassing the event composition. In contrast, the whole idea of Solicitor is to enable event composition for subscribers with real-time requirements.
Distribution aspects have not been emphasized. The only exception is that Gatziu [Gat94] has the possibility to specify that two event types should be part of the same transaction to enable event composition in a multi-database system.

Different coupling modes have been addressed in these prototypes, since they all have inherent transactions. In REACH (as mentioned earlier in Section 3.5 on p. 35), they address the lifespan of events [BZBW95] without going into detail about how to handle it. Also, in SAMOS Gatziu [Gat94] addresses how to specify composite event types stemming from different transactions. Invocation, scheduling, and delivery have not been addressed in these sources. Pruning is inherent in these systems, since event monitoring is performed within the scope of transactions.

10.12.3 Schwidersky’s Monitoring of Distributed Systems

Schwidersky [Sch96] addresses timestamping and different distributed protocols for event composition. These protocols, called synchronous and asynchronous, do not assume stable messages in contrast to what is assumed for Solicitor. Basically, the synchronous protocol is based on remote procedure calls, for example, if an event type is evaluated, then the physical nodes signalling occurrences of contributing event types are requested to validate that the currently evaluated event occurrences are stable. In the asynchronous protocol, there are no guarantees for correct event composition.

10.12.4 SMILE

SMILE is introduced by Jaeger (et al.) [Jae97, JO98] and is an active database system for situation monitoring in dynamic settings. In particular, parallel event composition and dynamic subscription are features of this architecture. Real-time issues are not emphasized. Distribution is discussed, for example, she uses timestamping with global and local parts for the ordering of event occurrences from different nodes. However, interoperability is not explicitly addressed. There is support for transactions, however, similarly to Solicitor, this support does not extend to subscribers that receive composite event occurrences outside the scope of transactions. Event composition processing is extensively addressed, in particular, parallel event composition. In contrast to this Ph.D. thesis, various forms of parallelism are investigated and a hybrid solution is proposed. In particular, the scheduling of event composition is briefly discussed, for example, the problem of
resolving disjunctions properly is addressed. The solution is that a node
processing disjunctions sends out probes to the nodes generating contribut-
ing event occurrences of the contributing event types. This is a more relaxed
and less general scheduling than is proposed to be used in Solicitor. This
more relaxed scheduling is suitable for high-performance systems, but for
real-time systems the value decreases with requirements of increased con-
trol over the situation. Similarly to Solicitor, event monitoring consists of
collection, composition, and delivery.

10.12.5 Event Pattern Language

The event pattern language (EPL) [MZ97] for deductive databases does not
address real-time or distribution aspects. Transactions are implicitly sup-
ported, since event composition is assumed to be performed within the scope
of a transaction. In terms of event composition, the kind of invocation and
delivery is not addressed, and the scheduling is data-driven. Declarative
event monitoring is supported.

10.12.6 GEM, Hermes, COBEA, X$$^2$$TS

GEM [MSS97], COBEA [MB98, Pie00], X$$^2$$TS [LCB99, CL00, CBB01, LT01],
Hermes [PB02, PS02] all address event composition in a distributed envi-
ronment where interoperability is important, since they aim for large-scale
heterogeneous distributed applications. In contrast, Solicitor aims at real-
time systems that may be large-scale heterogeneous distributed applications.
However, no provision for this is made in the current architecture of Solicitor.

GEM

The application area for GEM [MSS97] is network monitoring. Real-time
features have not been addressed. In terms of support for distribution, they
have considered relaxing the strict stability requirement of events by adding
a time window to each event type instead of that all event types have the
same time window as in stable messaging. Moreover, they have addressed
event monitoring in a situation where event occurrences may arrive out of
order. However, there is no convincing argument that this works in practice,
since only an example of the non-occurrence operator is considered where
they show it does not matter if events arrive out of order. The problem is
that actions may be taken depending on incorrect ordering and this is not
addressed in their work. There is no support for transactions. In terms of event composition processing, they address concurrent event composition, but the rest is not addressed. The event subscription is dynamic in terms of association of event type and subscriber.

It is possible to prune event occurrences from a connector set in an action. In contrast, Solicitor does not allow this. The major problem is to use these actions in an appropriate way, since event composition is difficult (at best) to understand [BMH99].

**COBEA**

COBEA [MB98] is a CORBA-based event monitoring middleware. A set of CORBA interfaces are defined together with an architecture for event monitoring. In particular, mediators and composite event servers are key components of this architecture that can be connected to each other. Real-time issues are not addressed. In terms of distribution, interoperability aspects are addressed. There is no support for transactions. Event composition processing is not addressed. The scheduling of event composition is data-driven. The event subscription is dynamic in terms of associating subscribers and event types. Declarative event monitoring is supported by a compiler [Pie00].

**X²TS**

The major issue addressed by Liebig et al. [LCB99] is that the $\delta_t$-precedence relation (in Section 3.2.2 on p. 32) is insufficient for large-scale distributed applications. Basically, a static clock granularity is impossible to maintain in many such systems and, therefore, they have proposed to use timestamps with clock precision intervals and they have also addressed how to maintain stability in event composition algorithms. Real-time issues are not addressed. Distribution is addressed in terms of time and order, interoperability. Also, Liebig et al. [CL00, LT01] have addressed the issue of transactions and event notifications. Their event composition is data-driven and event composition processing is addressed only in terms of stability. Subscription is dynamic.
10.12 Related Work

Hermes

Hermes [PS02, PB02] is an architecture where lessons learned from COBEA [MB98, Pie00] are addressed. In particular, event composition in mobile environments is addressed. For example, event composition and routing are addressed by Pietzuch et al. [PB02]. In contrast to COBEA [MB98], the coupling between different tasks performing event composition can be dynamically changed during run-time. Moreover, they define an XML-schema for propagating event occurrences over the network. Since mobile environments are prone to failures, fault-tolerance is addressed.

Real-time features are not addressed. Support for distribution is addressed, except that time and order is not emphasized. Transactions are not addressed. Event composition is implicitly invoked, scheduled according to a data-driven approach, and results are delivered immediately. Subscription is dynamic.

10.12.7 Monitoring using Real-Time Logic

Monitoring of real-time logic has been addressed in several sources [CJD91, JRR94, ML97a, ML97b, LMK98, LM99, LMY99]. Time constraints are inherent in any expression and, therefore, expiration can be expressed. Distribution aspects have been addressed by Jahanian et al. [JRR94]. Timely event monitoring has been addressed by Mok and Liu [ML97a, ML97b]. A unified approach of event composition and monitoring of real-time logic has been addressed by Liu et al. [LMK98, LM99, LMY99].

The architecture of Liu et al. [LMK98, LM99, LMY99] allows mapping of expressions in real-time logic and tasks performing event composition. However, the criticality issue has not been emphasized. With respect to distribution, nothing has been discussed except by Jahanian et al. [JRR94], who address time and order issues in particular. Interoperability issues have not been addressed.

Support for transactions are not stressed in any of these sources. Moreover, invocation, scheduling, and delivery have not been emphasized. At least two sources [LMY99, LM99] address support for declarative event monitoring by introducing a compiler.
10.12.8 Chronicle Recognition

Sources in Chronicle recognition (e.g., [DGG93, Dou96, Gha96]) do not fully address an event monitor architecture. Instead, they focus on composition processing. The available tool [CRS] leaves a lot to the programmer, for example, how to implement delivery of results, how to invoke event composition, how to optimize event composition by reducing duplication. Real-time issues have been addressed, since time constraints are an inherent optional part of Chronicle specifications. However, criticality and mapping of event monitoring functionality and tasks are left to the programmer. Neither support for distribution nor support for transactions has been addressed. Event composition processing is discussed, for example, how to handle duplication [DGG93]; however, this is, as mentioned earlier, something that the programmer has to implement. Therefore, declarative monitoring is only partially addressed in these sources.

10.12.9 Temporal Calculi for Communication Systems

The architecture defined by Bækgaard et al. [BG97] is coarse in comparison to the architecture proposed in this Ph.D. thesis. This coarse architecture only defines subscription. In contrast to the proposed architecture, their architecture allows dynamic event subscription whereas a static event subscription is assumed in the proposed architecture. In other words, only when a task is ready to receive an event occurrence does it request such an event in their architecture.

10.13 Summary

To summarize, we have presented a software architecture for event monitoring in embedded real-time systems that emphasizes significant issues such as mapping between event types and tasks, expiration management, criticality management. We advocate the use of compilers to enable increased programmer productivity by supporting declarative event monitoring.
Chapter 11

Memory Management of Event Composition

“Engineering is the practice of safe and economic application of the scientific laws governing the forces and materials of nature by means of organization, design and construction, for the general benefit of mankind.”

/S. E. Lindsay

One source of unpredictability is the memory management required for composite event management. To this end, a predictable and efficient solution to the problem of memory management is presented.

As mentioned in Section 2.1 on p. 12, event detection is the process of monitoring events (from the sources) and notifying event subscribers (e.g., application tasks) when events occur (depicted in Fig. 2.2 on p. 14). In this chapter, the important fact is that each component in Fig. 2.2 on p. 14 may reside in a separate address space.

The connector sets in Section 7.1 on p. 152 can be viewed as an optimization of event monitoring where intermediate results (composite event occurrences) of event composition are stored. These intermediate results are stored in a master copy owned by a monitoring task. That is, the event composition is viewed as processing a filtered event log, storing intermediate results representing the behaviors that may occur in the future (incompletely composed events), and signalling events when they occur [BMH99].
Memory Management of Event Composition

It is desirable to store the parameters of composite events in directed acyclic graphs (e.g., as depicted in Fig. 11.1), because many event specification languages are based on operator grammars (e.g., Snoop [CM94] and SAMOS [Gat94]).

For example, given the event history in Table 11.1, the table shows the content of the connector sets after each time step using the chronicle event context. The $\Gamma(\text{E}_1;\text{E}_2,[1,2])$ is represented by the pointer structure pattern depicted in Fig. 11.1.

### 11.1 Memory Management Problem

The memory management of event occurrences is a crucial part of composition, because the algorithms use allocation, deallocation, and copying of queued event occurrences. For example, given that we monitor for the event sequence $S_1$ that is specified as $\text{vehicleEnters}(\text{ptype: } \text{eid})\text{vehicleExits}$ (generated by the same vehicle entity identifier ($\text{eid}$)). Assume that $E1=\text{vehicleEnters}$ and $E2=\text{vehicleExits}$ in Fig. 11.1. Each time a vehicle enters, memory is allocated for an event occurrence of the type $\text{vehicleEnters}$ (e.g., $\Gamma(\text{E}_1,[1,1])$ in Table 11.1). When the same vehicle exits, memory is allocated for the $\text{vehicleExits}$ event ($\Gamma(\text{E}_2,[2,2])$) and for the event sequence $S_1$ ($\Gamma(\text{E}_1;\text{E}_2,[1,2])$). When the sequence is signaled to a subscriber, the al-

![Directed acyclic graph, representing an event occurrence of $E_1;E_2$ [Mel00]](image)

**Figure 11.1: Directed acyclic graph, representing an event occurrence of $E_1;E_2$ [Mel00]**

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event history</td>
<td>$\Gamma(\text{E}_1,[1,1])$</td>
<td>$\Gamma(\text{E}_1,[2,2])$</td>
<td>$\Gamma(\text{E}_2,[3,3])$</td>
</tr>
<tr>
<td>$Q_{E_1}(E_1;E_2)$</td>
<td>${\Gamma(\text{E}_1,[1,1])}$</td>
<td>${\Gamma(\text{E}_1,[1,1]), \Gamma(\text{E}_1,[2,2])}$</td>
<td>${\Gamma(\text{E}_1,[2,2])}$</td>
</tr>
<tr>
<td>$Q_{E_2}(E_1;E_2)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Q_{E_1E_2}(\text{recipient})$</td>
<td></td>
<td></td>
<td>${\Gamma(\text{E}_1;\text{E}_2,[1,2])}$</td>
</tr>
</tbody>
</table>

**Table 11.1: Event history example**
Memory management problem 247

located memory is copied to a buffer (a new address range) that is used for passing the event to a subscriber. Preferably, memory should be allocated, copied, and deallocated as few times as possible.

To handle different address ranges, it must be possible to copy from source to target address ranges without changing all pointers within the address range. For example, given that there are pointers referencing an element in an address range, these pointers must be relative to the start of this range. If we need to copy an address range containing a directed acyclic graph to another range, then the pointers must point to the correct position in the target memory and not the source memory. Self-referencing memory, that is, the possibility of maintaining pointers stored in the address range after copying the content to another range, is introduced to alleviate this problem.

The copying of a directed acyclic graph is referred to as cloning, because it performs a deep copy of the whole tree. For example, to signal the event sequence depicted in Fig. 11.1 on the facing page, it is necessary to traverse the graph from the top node while copying it to a buffer. By using self-referencing memory, it is possible to optimize cloning, because it is possible to make a raw copy of a self-referencing memory. To make this space efficient, the self-referencing memory must be sized to fit only the composite event (graph) in question. However, the master copy may contain several composite events and, hence, it is not space-efficient to copy it directly. Thus, the first cloning can be computationally more expensive than successive clonings of the same composite event. The reason is that, during the first cloning, the event monitor may have to construct a self-referencing memory containing the graph representing the composite event occurrence (that should be signaled). In contrast, in successive clonings it is possible to make a raw copy of the result of the first cloning. Moreover, lazy composition of parameters is desirable in soft real-time systems. For example, in network monitoring [LM99], the number of events signaled to the subscribers was significantly reduced as a result of event composition (up to 3–4 magnitudes).

The pointers to the event occurrences must be correct, that is, no dangling pointers should be left anywhere, and, preferably, memory leaks (memory that is not deallocated when it is no longer in use) should be avoided. The former should be guaranteed by the memory management, whereas the latter can only be completely supported if there is support for garbage collection in the execution environment.

To summarize, a generic approach suitable for various event composition algorithms is desirable. The approach needs to have bounded resource
requirements, avoid unnecessary memory copying, support correct programming by using abstract data types (e.g., safe pointers), and support self-referencing memory (for address ranges).

11.2 Self-Referencing Memory with Safe Pointers

The solution is to introduce a set of closely related abstract data types that encapsulate the required memory and pointer management. The memory management types provide: self-referencing memory, allocation and deallocation of memory for the event occurrences represented as nodes in the directed acyclic graph, and cloning of directed acyclic graphs into a separate address range for events that are sent to the subscribers. The design specifically addresses efficiency by making each important software construction as efficient as possible and predictability by not introducing unpredictable software constructions.

A pointer type is the only way to reach nodes in the representation of composite event occurrences. This partly ensures that we cannot get any dangling pointers. Further, the memory slots referenced by the pointers are not deallocated until the last pointer of a slot is destroyed. This is solved by using reference counters. Together, these two measures ensure that we do not get dangling pointers. However, we may still suffer from memory leaks, but this does not occur in a correct event composition algorithm.

11.2.1 Abstract Data Types

In Fig. 11.2 on the facing page, the abstract datatypes are depicted. Only vital operations are included. An event occurrence memory (EIMemory) consists of 1 or more event occurrence slots (EISlot). Each master copy as well as the field containing the event occurrence(s) in the messages carrying them is of the type EIMemory. The responsibility of EIMemory is to keep track of which slots are free, to give efficient access to a requested slot, and clone a requested (composite) event into a new EIMemory. Further, if the EIMemory is not a master copy, then it also must destroy itself when all pointers to it have been destroyed. The responsibility of EISlot is to keep track of how many pointers that are referencing it.

Each EISlot contains an event occurrence (EI). EISlot and EI are distinguished to separate the responsibilities. A node in the representation of an event occurrence is one out of the following categories: primitive (PEI),
11.2 Self-Referencing Memory with Safe Pointers

Figure 11.2: Abstract data types for event occurrences (based on [Mel00])

Unary composite (UCEI), or binary composite (BCEI). These categories partition event types depending on if it is primitive or composite, and if it is a composite whether the result of the operator is unary or binary. If the interval of the composite event occurrence is only dependent on the result of one contributing event type of an event operator, then the result is unary. This is the case for disjunction, aperiodic, and terminator operator. Sequence, conjunction, and the non-occurrence event operators all produce a binary result.

The pointers (Ptr2EI) are associated to a specific EIMemory. A pointer is active if it refers to an EISlot and it is associated if it belongs to an EIMemory. An active pointer can neither be associated with another EIMemory nor
be allocated to another slot without being destroyed. A pointer in one EIMemory can point to an event occurrence stored in another EIMemory; for example, if a composite event occurrence is signalled from one monitoring task to another, then the master copy of the receiving task only maintains a pointer to the EIMemory of the message containing the composite event occurrence. The 'create' method is overloaded for each event category. The 'create' call allocates an EISlot and calls the appropriate 'create' method of PEI, UCEI, or BCEI.

11.2.2 Self-Referencing Memory

There are two ways to achieve the self-referencing property of the EIMemory. Firstly, each memory slot (EISlot) is considered to be of the same size and stored in an array. This allows efficient memory allocation and deallocation without external fragmentation at the cost of internal fragmentation. Internally, each pointer keeps an index to this array. Hence, it does not matter in which address range the event occurrence memory resides. Secondly, to handle user parameters on arbitrary addresses the raw address is stored in PEI. This raw address is relative to the start of the event occurrence memory address range. This last method allows us to perform lazy parameter composition of user parameters. Moreover, to ensure that the pointers to primitive event occurrences are correct, only the start of the EIMemory itself (stored in the startAddress parameter) must be maintained.

Given this, it is possible to copy the self-referencing memory to another address range and, hence, it is only necessary to change the startAddress parameter. This parameter is used to obtain the user parameters by adding it to the relative raw pointer to obtain the actual raw pointer.

11.2.3 Safe Pointers

The pointers are made safe by guaranteeing that each time a pointer is copied (as in assignment or as sending it as a parameter to a function call) the reference count is increased. Each time an automatic variable on the program stack is removed, the reference count is decreased if the pointer is active. Moreover, each time a pointer is destroyed the reference count is decreased. When the reference count is 0, the memory slot is deallocated, which may introduce a bounded cascade of memory deallocations. If a deallocation of a slot in an EIMemory (EIMemory A) that references a slot in another EIMemory (EIMemory B) causes a cascaded deallocation of all slots
in EIMemory B, then the buffer containing this EIMemory B is deallocated by using the operating system services for memory management.

11.2.4 Optimizing For Cloning

The pointer structure is optimized for an event composition algorithm, in particular for the first cloning. The first cloning from the master copy to a clone involves two steps: (i) to obtain the space requirements and (ii) to construct the clone in a self-referencing memory based on the master copy. By keeping track of the size requirements in each node in the representation of event occurrences, it is not necessary to traverse the graph during step (i). This is done by recording the slot requirements of the constituents, that is, one for the slot of the composite itself, and sum of the size of user parameters in each constituent. For example, $E_1$ and $E_2$ in Fig. 11.1 on p. 246 require 1 slot and, for example, 10 bytes each for user parameters, then 3 slots and 20 bytes is recorded in the node representing $S_1$. Memory waste is built in, because nodes may be shared in a composite event. However, as mentioned in Section 8.4.1 on p. 192, common subexpressions (which result in shared event occurrences) can be useless in practice.

11.2.5 Lazy Composition of User Parameters

Performance can be improved significantly by delaying the actual parameter copying until an event is signaled to its subscriber, which is the case when only few (primitive) event occurrences are actually part of signalled composite event occurrences. Lazy composition of user parameters implies keeping primitive events in their buffers as they arrive to the event monitor. The user parameters are copied into a clone when the (composite) event is signaled. This copying of user parameters from primitive event occurrences in a master copy into a clone, allows creation of new clones by making a raw copy of the first clone.

11.2.6 Aggregation for Correct Signalling

One important part is to aggregate event occurrences of different event types that are to be sent to the same subscriber. In particular (as mentioned in Section 4.9.2 on p. 77), for correct results in explicitly invoked event composition or delayed delivery in implicit invocation of event composition, it is necessary to deliver all results in an atomic operation rather than several
operations (messages). Further, this measure can decrease the number of messages sent via inter-task subscription. In particular, if event composition is explicitly invoked or if there are several composite event occurrences with the same terminating event type, then this measure can significantly decrease inter-task subscription communication.

11.3 Realization in C++

The design was implemented in C++, where all methods except destructors, allocate(), deallocate(), and clone() were declared inline, because they either are recursive or there is no significant gain in making them inline. Regarding the mapping of the abstract data types into C++, the aggregate between EISlot and EI cannot be properly declared in C++. The reason is that arrays are ill-behaved with respect to derived classes in C++ [Lip96]. For example, if we declare an array of EI, it is not possible for a compiler to find the right size and the right constructor to use. The solution is to declare the array to be of the largest subclass of EI (BCEI in this case) and Ptr2EI is responsible for calling the correct construction method. Moreover, to guarantee safe pointers, Ptr2EI has the constructor, destructor, and the assignment operator conditioned to handle reference counting.

11.4 Empirical Study

Three experiments have been performed. All of them only consider the memory management of the complete architecture in Fig. 10.3 on p. 225; therefore, the results of these experiments show the potential of memory management while ignoring the impact of all other layers. Experiments dealing with the whole architecture are addressed in Ch. 12.

The first studied the effects of inlining and optimization of memory management. The second considered the potential gain of successive clonings. The third experiment emphasized the performance gain in terms of throughput with respect to event composition using ordinary memory management. In these experiments, the programs were compiled with both GNU C++ (version 2.8.1) and SPARC C++ (version 3.0.1). The experiments were executed on a 300 MHz single-processor Ultra SPARC10. The clock routine in UNIX was used to obtain the throughput of the simulation. The precision of the clock is 1 ms and, given that each simulation was executed 100000 times, the measurement error is 0.01 $\mu$s.
In the first experiment, as a result of inlining, the memory management code executed approximately four times faster than its non-optimized counterpart without inline functions. The first cloning of an event such as $Not(\ e_e, E_1; E_2, E_d)^1$ composite event took on average 30 $\mu$s for the best optimization (level 3) and on average 82 $\mu$s without optimization with inlining. Without inlining, the averages were 52 $\mu$s and 120 $\mu$s respectively. The primitive events contained 2 bytes of user data.

The second experiment was conducted under the same conditions as the first experiment. The speed-up between the first and successive clonings (to clone the clone) was considered. The result is that the speed-up is approximately 1.3 to 2 times for the $Not(\ e_e, E_1; E_2, E_d)$ event. The more complex the specification is, the more is gained from the proposed method.

In the third experiment, the composition of the sequence in Fig. 11.1 on p. 246 was considered. The baseline version used ordinary memory management and copied the event occurrences for each step. The tested version used the proposed memory management. The number of recipients of the sequence and the number of events queued at the operator were varied. In terms of throughput, the proposed method is approximately 20% faster than the baseline. The investigated optimization for successive clonings was not addressed. Also, the proposed method seems to scale better than the baseline, but this must be more thoroughly investigated.

**11.5 Applicability**

This approach is directly applicable to event monitoring for Snoop, and in SAMOS, since these are based on operator grammars suitable for storing composite events in directed acyclic graphs. For the other approaches such as ODE (e.g., [GJM93]), monitoring of real-time logic (e.g., [LMK98]), and temporal calculi for communicating systems [BG97], this approach is directly applicable to a subset of the specifications. However, it is conjectured that the efficiency of our approach renders more expressive event composition approaches (based on more complex grammars) less useful for large-scale systems.

---

$^1\ e_e$ is the event enabler, and $E_d$ is the event disabler.
The computations have bounded resource requirements as a result of the following reasons:

1. There is a statically defined number of slots in an EIMemory, that is, a directed acyclic graph representing the composite event is limited. To try to allocate more than the maximum number raises an exception.

2. Each event specification consists of a fixed number of contributing event types and the number of event specification monitored by a monitor is fixed. No specification introduces a cycle.

3. Finally, by using expiration and ensuring that there is minimum interarrival time between event occurrences, the size of connector sets is bounded as discussed in Ch. 8.

11.6 Related Work

In contrast to the work in Snoop, SAMOS, ODE, monitoring of real-time logic expressions, and TCCS, this work has focused on a predictable and efficient memory management of composite events.

In contrast to the problem of marshalling in remote procedure calls [BN84] and CORBA (e.g., [CHY+98]), this does not focus on heterogeneous architectures.

In contrast to safe pointers (in C++ literature), this work also adds self-referencing memory.

In contrast to the article [Mel00], there are no trinary composite event instances (TCEI). The reason is that no event operator results in such an event instance. Moreover, if there were such operators, then the BCEI can be used to model any size instances by linking several BCEI’s together.

Further, this chapter also addresses how to treat memory for signalled composite event occurrences, an issue that was not covered by the article [Mel00].

11.7 Future Work

Two extensions are of interest: (i) It is desirable to extend this method to cyclic graphs. For example, the time constraint monitoring [LM99] is dependent on the efficiency of copying a potentially cyclic graph. (ii) As
11.7 Future Work

mentioned in Sections 10.6.2 to 10.6.3 on p. 235, intelligent pruning of event occurrences can save computational resources and it is desirable to check how this affects memory management.

The performance gain, for example, in execution time, and cost, for example, in increased object code size, is desirable to study more thoroughly in a real environment such as the DeeDS prototype [AHE+96] using the event specification language Solicitor. In addition, the queue management of event composition can probably be handled in a similar way to improve performance even further.
Part IV

Results
This part contains empirical studies (in Ch. 12), an overall discussion (in Ch. 13) and comparison to related work (in Ch. 14). More detailed discussions concerning results and comparisons to related work are found in each chapter in Part II and Part III.
Chapter 12

Empirical Studies

“God does not care about our mathematical difficulties. He integrates empirically.”

/Albert Einstein

“The propositions of mathematics have, therefore, the same unquestionable certainty which is typical of such propositions as ’All bachelors are unmarried,’ but they also share the complete lack of empirical content which is associated with that certainty: The propositions of mathematics are devoid of all factual content; they convey no information whatever on any empirical subject matter.”

/Carl G. Hempel

In this chapter, the theoretical results addressed in Ch. 6 to Ch. 9 are validated. In particular, the complexity classes of different contexts are addressed. The results of these experiments corroborate the results of the time complexity investigation based on a theoretical model.

12.1 Foundation of the Experiments

To make the results of the experiments credible, this section defines the target platform, the overall design of the experiment, the variables affecting the
experiment, management of factors of the experimental resources, and the
type of analysis that can be performed on the collected data in accordance to
the requirements of a formal experiment as defined by Fenton and Pfleeger
[FP96].

12.1.1 Experimental Target Platform

The experimental unit is an implementation (in C++) of an event monitor
for the Solicitor event specification language following our software architec-
ture for embedded systems defined in Part III within the DeeDS (Distrib-
uted Active Real-Time Database System) [AHE+96]. This event monitor
is a separate software component and it can be used without the database
management system.

Task management, memory management, and message passing used by
the event monitor is provided by the DeeDS operating system interface that
is an adapter ported to OSE Delta, Solaris, and Linux. In this experiment,
the underlying operating system is the OSE Delta 3.2 kernel running in emu-
lation mode on a host target with full error detection and system debugging
turned on. The host target is a Ultra SPARC10 (300 MHz) running So-
laris 2.6. To compile the C and C++ code, the GNU compiler version 2.8.1
was used. All the experiments were performed with debugging information
compiled into the monitor to enable location of faults.

12.1.2 Overall Design of Experiment

The experiments are executed by automated scripts written in PERL. Each
experiment consists of a set of trials: one trial for each combination of values
assigned to variables that are interesting. Each trial does the following:

- generates a monitor specification in Solicitor, an event producer task
  signalling primitive event occurrences, and an event consumer task
  receiving event occurrences

- compiles a trial program

- executes a trial program repeatedly

- collects results and stores them in a file
Each execution consists of three phases: initialization of the trial, the actual trial, and reporting the results to the script executing the trial program. During the initialization, the event producer signals a set of initial event occurrences that does not result in any composite event occurrences. During the actual trial, a set of terminator occurrences are signalled to the event monitor. The resulting composite event occurrences are received by the event consumer that records how many occurrences it has received as well as simulates usage by iterating over these received event occurrences. When the expected number of composite event occurrences has been received, the trial is completed and the duration per occurrence is reported back to the PERL script.

There are several types of problems causing experimental errors that need to be managed according to Fenton et al. [FP96, p. 131]: (i) errors of experimentation, (ii) errors of observation, (iii) errors of measurement, and (iv) the variation in experimental resources. Problem (i) is managed by using appropriate tools such as compilers as well as inspections, testing of the code of the experiments, and configuring the experimental resources to fit the experiment alone. For example, a resource such as the maximum number of event occurrences that an EIMemory (defined in Section 11.2.1 on p. 248) can hold is sized to fit the experiment, since this strategy enables early error detection of memory leaks. Problem (ii) is managed by reducing the intrusion of the instrumentation (for experimental purposes) so that its performance overhead is insignificant with respect to overhead of the tested code as well as ensuring that the intrusion is comparable for all experiments. This is addressed in the next paragraph. Management of problem (iii) and (iv) are discussed in Section 12.1.4 on p. 265.

The measurement of duration in terms of execution time is done by reading the used execution time after the initialization step of the experiment has been completed and when the event consumer has received the last composite event occurrence. The period between these two timestamps is the total duration it takes to signal, process, and receive composite event occurrences based on the terminator event occurrences in the trial. By dividing this total duration with the number of terminator event occurrences, an average processing time for each terminator event occurrence is computed. The reason why individual measurements for each terminator event occurrence are not observed and recorded is that the granularity of the execution time clock is larger than the duration of each individual processing of a terminator occurrence. To measure the total duration and divide by the number of
terminator event occurrences is a viable solution, since earlier experiments on an embedded system with a hardware-timer with finer granularity (1 µs) reported in the licentiate thesis [Mel98a] strongly indicated that the variance of duration of event composition is small.

### 12.1.3 Variables

The following variables (factors) are controlled in the experiments (where all variables denoting sizes are derived from Section 9.2 on p. 201):

**Event context:**

The event context can be set to recent, chronicle and general\(^1\).

**Size of the filtered event log (per primitive event type):**

\((|G|, \hat{g})^2\) This size is approximated with the number of terminator occurrences signalled after the initialization phase of a trial, since the number of terminator occurrences is assumed to be proportional to the size of the filtered event log.

**Event type size:**

\((|E|, \hat{e})^2\) The size of event types is based on the number of operators in the composite event specification.

**Parameter size:**

\((|P|, \hat{p})^2\) The size of the user parameters is the number of integer parameters in each primitive event type in the specification. All event types contain system dependent parameters such as the time span of occurrence.

**Pointer type:**

The pointer type can be safe or unsafe.

**Repetition:**

The number of times the execution should be repeated to avoid basing results on outliers.

The scheduling of event composition is control-driven, invocation of event composition is explicit (initiated once for each signalled event occurrence during the actual trial), and delivery of the resulting event occurrences is delayed.

---

\(^1\) In general context, only for the disjunction, sequence, aperiodic, and relative temporal operator.

\(^2\) Variable is defined in Section 9.2 on p. 201.
12.1 Foundation of the Experiments

12.1.4 Managing Factors of Experimental Resources

There are several factors affecting experimental resources in computer science. These factors must be managed to make the experiments credible. Factors that cannot be controlled are managed by only measuring execution time (in contrast to elapsed time) and repeating the trials so that statistical analysis can be used. Statistical analysis can be used since only relative measurements are of interest. For example, in validation of time complexity it is desirable to check how the duration of event composition increases with respect to the size of the event type. That is, it is only interesting to compare the different durations resulting from different event type sizes with each other rather than with some absolute scale. A necessary requirement is that the worst-case execution time and the average-case execution time scales the same way. This requirement is conjectured to hold, since all algorithms used to realize Solicitor are of linear or better time complexity.

The factors can be broadly divided into hardware and software factors. The hardware factors are managed by repeating trials to statistically manage the result of the experiment, since these factors are hard to control and only relative measurements are necessary. A more detailed presentation of the factors are as follows:

- Hardware-related factors:
  - Processor-related factors:
    - Many recent processors have at least one pipeline and at least one level of memory cache. Both of these factors are hard to control. Moreover, even if it is possible to control them, it is difficult to do so in a realistic way. For example, according to Engblom [Eng02, pp. 97–98] set-associative memory caches use random replacement algorithms of parts of the caches only enabling statistical models for worst-case execution time.
    - In addition, (asynchronous) interrupts affect the behavior of a processor. Interrupts are due to that other hardware devices require the processor to service them. For example, when an ethernet packet arrives the processor is notified by using an interrupt that there is a packet that needs to be processed. An alternative to employing repetition of trials to manage interrupts is to turn off all possible interrupts.
  - Bus-related factors:
    - Direct memory access transfers can be used to transfer large amounts of data over the bus while other memory accesses take
place. However, there is an overhead penalty for other memory accesses due to direct memory access transfers. These transfers are initiated by interrupt service routines. For example, the network packet is moved from a (limited) hardware memory area to another part of the memory for further processing.

**Memory-related factors:**
Main-memory access is often optimized for aligned data. Aligned data require that the physical memory address is divisible by, for example, 4. If data is unaligned and it is allowed to be unaligned, memory access time may increase significantly.

Also, in modern operating systems, logical address spaces are used to separate processes from each other. There is an overhead cost associated with mapping logical addresses to physical addresses, since this involves translation-lookahead buffer accesses [Eng02]. This translocation buffer is located in main memory so that it a memory access to may require several memory accesses before finding the physical memory in question.

- **Software related factors:**

**Operating system related factors:**
Processor, task, and memory management as well as interprocess communication are basic features of operating systems. In particular, the scheduling of tasks on resources such as processors, communication channels etc., can affect the experiment. To deal with this, the experiment is designed in such a way as to minimize the effect of scheduling, for example, by avoiding race conditions.

Paging and swapping are used to provide the illusion of more main-memory than there actually is. If the memory page to be accessed is not in main-memory, then there is an overhead penalty for loading the memory page from a disk into main memory. It is possible to lock an application into main memory. However, this is only available for the superuser and, therefore, it is unavailable for our experiments. Moreover, some experimental configurations require more main memory than the available main memory on the target system. However, the trends can still be considered, since the effect of paging is assumed to be evenly spread through a trial.
12.2 Validation of Algorithmic Time Complexity

Run-time execution environment related factors:
This is closely related to the operating system, but deals with language specific issues such as exception handling, reflection services (if any) etc. Basically, none of these services are used during normal processing. Exceptions may be raised, but these are for development purposes denoting that some assertion does not hold. In such a case, the fault has to be managed, for example, by removing it.

12.1.5 Enabled Analysis of Collected Data

It is possible to use all arithmetic operators according to the representational theory mentioned by Fenton et al. [FP96, sections 2.3–2.4], since the duration is based on a ratio scale. For curve fitting, Tukey’s ladder [FP96, pp. 216–218] was used with the help of MATLAB.

12.2 Validation of Algorithmic Time Complexity

It is desirable to validate the accuracy of the theoretical model of event composition (in Ch. 6 to Ch. 9) with respect to a real implementation, since event composition is a complex task performing different basic operations. Experiments are chosen to validate this accuracy, since it is difficult to analyze the code; for example, how much will memory management contribute to the complexity in comparison to the control-driven scheduling of event composition?

In contrast to, for example, sorting algorithms whose basis is the number of comparisons necessary to sort an array of elements, event composition performs lookups, comparison, memory allocation etc., which all contribute to the computational cost (and, hence, asymptotic efficiency). This fact is addressed in Ch. 9.

This section starts with a demonstration that the distribution of average response times for different configurations of event composition is close to parametric (i.e., a bell-shaped distribution). Then, we address the dependency between maximum, average, and minimum response time and the variables event type size, size of the filtered event log, and parameter size.
12.2.1 Distribution of Response Times

The distributions are close to being parametric as can be seen in Appendix D; this implies that (i) it is important to repeat the experiment to avoid basing the results on outliers as well as (ii) we can use statistical analysis based on parametric distributions (e.g., students t-test [FP96]).

The distribution of response times was measured for the following combinations of configurations:

- event context: recent, chronicle
- event type size ($|E|$): 1, 10, 50
- event history size (per primitive event type) ($|G|$): 10, 100, 1000,
- event parameter size: 0
- pointer type: safe 10000
- requested repetition: 100

94 out of 100 trials (repeated results) were used, since in the worst case 6 trials failed due to that the OSE emulation kernel fails to start at all. Also, due to resource limitations the combination of 50 event operands and 10000 event occurrences failed for all repetitions (this configuration requires the system to maintain 500000 primitive event occurrences).

12.2.2 Dependency on Event Type Size

Recent and Chronicle Event Contexts

The effect of scaling the event type size for 1, 2, 3, . . . , 29, 30 event operators was measured for the following configurations:

- event context: recent, chronicle
- event history size (per primitive event type) ($|G|$): 1000, 5000, 10000
- event parameter size ($|P|$): 0
- pointer type: safe
- requested repetition: 10
The collected data supports the theoretical results in Figs E.1 to E.3 on pp. 384–385 and Figs E.4 to E.6 on pp. 385–386 that recent and chronicle context scale linearly with respect to the size of the event type.

There is a high correlation (≥ 0.95) between the coefficients of a linear curve for each configuration. Also, the fitness value of a linear curve is good for each configuration.

Therefore, this experiment supports the theory that the algorithmic time complexity of event composition scales linearly with respect to the size of the event expression regardless of event history size for recent and chronicle event contexts.

**General Event Context**

![Graph showing response time dependent on type size for general event context](image)

**Figure 12.1**: Response time dependent on type size for general event context

As can be seen in Fig. 12.1, the experiments support that the response time scales exponentially with respect to the size of the event type. More exp-
Table 12.1: History variation for different $|E|$ in general event context

| $|E|$ | History size variation |
|------|------------------------|
| 1    | 1, 2, 3, ..., 43, 44, 100, 200, 300, ..., 1900, 2000 |
| 2    | 1, 2, 3, ..., 294, 295 |
| 3    | 1, 2, 3, ..., 43, 44 |
| 4    | 1, 2, 3, ..., 15, 16 |

Experiments could be conducted, however, the results are deemed to be sufficient to demonstrate how event composition in general event context behaves.

The type size varied from 1 to 4 and the history size was set to 5, 10, or 15. The reasons for these, in comparison to other event contexts, small values on configuration parameters are: (i) the effect of general event context is visible at these values and (ii) the experiments quickly run out of memory. For example, for 4 operators with 15 primitive event occurrences for each initiator, then it is necessary to store $\sum_{i=1}^{4} 15^i + 4 \times 15 = 54300$ event occurrences while processing a single terminator since delivery is delayed. Additionally, the parameter size was 0, the pointer type was safe, and the number of requested repetitions was 10.

12.2.3 Dependency of the Size of the Filtered Event Log

The effect of scaling the history size for 10, 20, 30, ..., 990, 1000 and, finally, 1100, 1200, ..., 9900, 10000 event occurrences per primitive event type was measured for the following configurations:

- event context: recent, chronicle
- event parameter size: 0
- pointer type: safe
- requested repetitions: 10

Fig. 12.2 on the facing page does not suggest that there is a relation between the algorithmic time complexity and the history size for chronicle event context. Likewise, Fig. 12.3 on p. 272 does not suggest this for recent event context. In contrast, as predicted, general context scales polynomially with respect to the history size as can be seen in Fig. 12.4 on p. 273. N.B.,
the result of $|E| = 1$ is not visible due to the scale of the diagram, that is, it cannot be distinguished from the x-axis.

In the experiments for recent and chronicle event context, the history size varied between 10 to 10000 event occurrences for each primitive event type. This was performed for event type sizes of 1, 10, 20, and 30 sequence operands. The parameter size was set to zero implying that no parameter (except the event type) is carried with each event occurrence.

In the experiments for general event context, the history size varied according to Table 12.1 on the facing page. The allocated memory sizes for the configuration were exhausted for larger history sizes except for $|E| = 1$. It would be possible to reconfigure the experiment, but the tried configurations are considered to be sufficient to demonstrate the behavior of general event context.
12.2.4 Dependency on Parameter Size

The effect of scaling the parameter size for 0, 10, \ldots, 90, 100 and additionally for 200, 300, \ldots, 9900, 1000 integers per primitive event type was measured for the following configurations:

- event context: recent
- event type size: 10
- event history size (per primitive event type) (|G|): 1000
- pointer type: unsafe and safe
- requested repetition: 10

The experiments for both unsafe and safe pointers (shown in Fig. 12.5 on p. 274 and Fig. 12.6 on p. 275) both corroborate the theory that event
composition is of linear time complexity with respect to the parameter size $\hat{p}$. Further, event composition based on unsafe pointers Fig. 12.5 on the following page has a higher overhead and scale worse than event composition based on safe pointers (cf. Ch. 9).

The variation of parameter size for unsafe pointers does not change the behavior significantly for normal ranges of parameter sizes as was expected from the initial tests in Section 11.4 on p. 252. The reason can be found by profiling the experiment. Given $|E| = 10$ and $|G| = 1000$ only 5 percent of the time is spent in copying event occurrences for constructing new composite event occurrences for unsafe pointers. It is conjectured that the design and implementation of event monitoring can be further optimized. For example, how appropriate is the current implementation of scheduling of event composition that only evaluate event types that can result in event occurrences? It can be more efficient to evaluate all event types regardless of whether they can result in an event occurrences or not and thereby reducing the overhead of scheduling while improving the total performance.

Figure 12.4: Response time dependency on history size in general context
12.3 Discussion

This is a short discussion concerning the relevance of the parameter choices as well as the accuracy of the results.

12.3.1 Relevance of Parameter Choices

The most important variable is the size of the filtered event log, since it may contain several thousands of event occurrences even for small applications. The experiments have ranged up to 10000 event occurrences per primitive event type. To give an example of how long time we can keep event occurrences in the filtered event log before they have to be pruned due to expiration, consider the axiom: ∀E (\text{prim}(E) \Rightarrow \lceil \text{expire}(E)/T_{int} \rceil + 1 = 10000) (cf. Ch. 8). For example, by using the equations in the introduction of Ch. 8, \( T_{int} = 0.1s \) and \( ∀E (\text{prim}(E) \Rightarrow \text{expire}(E) = 99990s = 27.775h) \), or \( T_{int} = 0.1ms \) and \( ∀E (\text{prim}(E) \Rightarrow \text{expire}(E) = 99.990s = 1m39s) \). Obviously, a larger number of event occurrences can be valid to test allowing us to keep a longer history,
but our experiments are performed in a range that is assumed to be valid for many applications.

Due to the complexity of understanding event expressions [BMH99], event expressions with more than 3 event operators are considered unlikely. However, the experiments have ranged up to 30 operators, that is, this range is 10 times more than expected. Therefore, it is conjectured that the experiments hold for current and future applications with respect to this parameter.

The parameter size has been chosen between 0 to 1000 integers (4 bytes per integer), where the expected value for an application corresponds to 0 to 20 integers. This is based on how many parameters there can be in a normal function, procedure, or method call. Therefore, the experiments are conjectured to hold for current and future application with respect to this parameter.
12.3.2 Accuracy of Results

Since these experiments were also the final tests of the event monitor for DeeDS, they were performed on an emulated operating system on a development target (Solaris operating system on a SPARC). The accuracy is deemed to be sufficient to support or reject the theory with respect to the classes of algorithmic time complexity for each event context. However, the results cannot be used to derive accurate cost models that are representative for real targets.

A greater accuracy can be obtained by performing the experiments on the host target as was done in the licentiate thesis [Mel98a]. However, it is deemed that these more time-consuming experiments on the host target was unnecessary to corroborate the complexity classes from Ch. 9. On the host target used in the licentiate thesis, it is possible to have a greater control of hardware factors as well as software factors than on the chosen target. It is even possible to achieve conservative pessimistic worst-case execution time measurements, since it is possible to turn off the caches as well as some pipeline functionality.

12.3.3 Comparison to Other Experiments in this Work

In contrast to the experiments reported in Section 11.4 on p. 252 that use implicit invocation and data-driven scheduling, the experiments in this chapter employ explicit invocation and control-driven scheduling.

12.4 Summary

To summarize, the experiments corroborate the significant algorithmic time complexity classes derived from the theoretical model of event composition. The configurations are based on relevant parameters for applications.
Chapter 13

Discussion

“History is the version of past events that people have decided to agree on.”
/Napoleon Bonaparte

In this chapter, we address an evaluation of suitability and usefulness of event composition in real-time systems, formalization of event composition processing, issues concerning order and time, performance considerations, support for real-time systems, and application examples of event monitoring.

13.1 Evaluation of Event Composition in Real-Time Systems

Suitability of event composition for real-time systems is how well event monitoring can be part of a task load with time constraints while the real-time system managing this task load achieves timeliness. In contrast, the usefulness of event composition addresses the added value of using event composition compared to alternative methods such as finite state automata.
13.1.1 Suitability: Achieving Timeliness

As discussed in Section 3.1.1 on p. 27, the necessary and sufficient conditions for timeliness are defined as predictability and sufficient efficiency of tasks (executing algorithms). As has been shown in Part II, predictability of event composition can be ensured by using expiration and sporadic events.

In contrast to predictability, sufficient efficiency is relative to the application requirements as well as the available resources. Locke [Loc96] has reported time constraints based on his experience from various industrial applications: air traffic control, aircraft mission control, spacecraft control, training simulation, and process control. The time constraints (called latencies by Locke) for updating a database is shown in Table 13.1, where the database maintains the state of the controlled process (e.g., an aircraft) that an application controls. By comparing these time constraints with the results from the empirical study (in Ch. 12), it is possible to discuss in which applications event monitoring can be used. N.B. the results from the empirical study are not worst-case execution times, but maximum measured execution times; however, the results give us at least an idea of which applications event composition is suitable for.

<table>
<thead>
<tr>
<th>Application</th>
<th># entities</th>
<th>Time constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
</tr>
<tr>
<td>Air traffic control</td>
<td>20000</td>
<td>0.5ms</td>
</tr>
<tr>
<td>Aircraft mission control</td>
<td>3000</td>
<td>0.05ms</td>
</tr>
<tr>
<td>Spacecraft control</td>
<td>5000</td>
<td>0.05ms</td>
</tr>
<tr>
<td>Training simulation</td>
<td>100000</td>
<td>0.5ms</td>
</tr>
<tr>
<td>Process control</td>
<td>unknown</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Table 13.1: Time constraints of updates based on Locke’s work [Loc96]

To address how well event composition can meet the time constraints by Locke, assume that event monitoring uses a fraction \( u \) of the processor. Given that the response time of event composition is \( r \) when \( u = 1 \), then the function \( rt(u) = r/u \) defines an approximate response time of event composition with respect to processor usage \( u \). N.B., this function ignores jitter. Moreover, assume that all event types are of size 10 or fewer operators in chronicle or recent event context. Actually, as mentioned in Section 12.3.1 on p. 274, we do not expect larger event expressions than of size 3 operator due to the difficulty of understanding the more complex event composition.
as discussed by Berndtsson et al. [BMH99]. Additionally, assume for this discussion that the maximum time constraint is equal to or less than the expiration time of event types.

![Figure 13.1: Suitability of event composition for process control](image)

Finally, assume that the minimum interarrival time of event occurrences is 10000 times faster than the expiration time, that is, $|G|=10000^1$. We need to store up to $|G|$ event occurrences per event type.

From our experiment, the maximum measured execution time is 6ms for $|G|=10000$ and $|E|=10$ (and 4ms for $|E|=1$) as depicted in Fig. D.8 on p. 375 (and Fig. D.4 on p. 373) (for chronicle context) and Fig. D.19 on p. 381 (and Fig. D.15 on p. 379) (for recent context); these figures depict the distribution of experiments (cf. Section 12.2.1 on p. 268), which are used because the results from the distribution experiments are based on more trials than the other addressed experiments. Based on these numbers, Fig. 13.1 indicates that event composition can be used for process control, where we can handle $1/rt(u)$ event occurrences per second (e.g., at $u=0.2$ and $|E|=1$ we can handle an event occurrence every 20ms or 50 event occurrences per second). In contrast, Fig. 13.2 on the following page does

---

1As mentioned in Table 9.1 on p. 201, the variable $|G|$ is defined as the number of event occurrences per event type.
not exclude that event composition can be used for air traffic control (and training simulation) if there is a dedicated processor for event monitoring. Given a dedicated processor for $|E|=1$, we can handle approximately an event occurrence every 4ms or 250 events occurrences per second.

The conjecture concerning air traffic control (and training simulation) needs to be pursued further, since Fig. 13.2 does not indicate as clearly as Fig. 13.1 on the previous page that event monitoring can be used. There are two important issues: (i) we do not compare the worst-case execution time of event composition with the application requirements and the results of the experiments are derived for an old platform (cf. Section 12.1.1 on p. 262), and (ii) the measured response time includes overhead such as polling (which usually is unacceptable).

We argue that event composition can be used for air traffic control and training simulation. Concerning issue (i), the worst-case execution time is conjectured to be a magnitude larger than measured. However, with hardware that is a magnitude faster than the hardware used in the experiments, this conjecture concerning air traffic control (and training simulation) can hold. Concerning issue (ii), the experiments (cf. Section 12.1 on p. 261, in
13.1 Evaluation of Event Composition in Real-Time Systems

particular Section 12.1.2 on p. 262) are based on an emulated operating system that uses a significant part of the bandwidth of the processor for polling and includes simulation of usage (by iterating over the received composite event occurrences); thus, the actual end-to-end (average) response times of event monitoring are less than the measured response times.

13.1.2 Usefulness: Alternative Methods

Event composition is useful in, at least, two major types of applications: (i) applications (i.e., the controlling tasks) where there is not a one-to-one relationship between states in the application and states of the controlled process, and (ii) applications where composite event specification reduces the size of system specifications.

To explain what the one-to-one relationship between states means, consider the following example: In a graphical user interface, there is usually a one-to-one relationship between the state presented by the interface (the controlled process) and the state of the application. When the user performs something, for example, clicks a button, the state changes in the interface as well as in the application. Therefore, event composition is unnecessary for graphical user interfaces from the perspective represented by application type (i). In contrast, in a database the application can submit a query and while this query is being processed by the database management system, there is not a one-to-one relationship between the application and the currently evaluated records in the database. Instead, the database management system maintains this one-to-one relationship between its processing of the query and the records that are being processed. When the query has been processed, the application regains the control and has a one-to-one relationship between its state and the result of the query.

Concerning application type (i): In such applications, different lines of reasoning are useful (cf. Section 2.3 on p. 22), since we need to speculate what is happening in the controlled process. In event composition, lines of reasoning are represented by composite event occurrences. For example, will we receive a response to a remote procedure call request over a network? In this case, we need to speculate that either we get a response or we do not get a response. Moreover, we may receive multiple responses as a result of one request, if diffusion is used.

---

2 The concept “controlled process” is used in a broad sense for the discussion in this section, not only process control.
Two typical type (i) applications, active databases and network monitoring, are addressed in the following paragraphs. Application examples (for active databases) are found in Section 13.6 on p. 291.

In active databases, the reason that they belong to type (i) is separation of concerns. *Separation of concerns* is defined here as the strategy to separate functionality into different software components to achieve some quality attribute. For example, integrity rules (if an integrity rule is violated, then the action is to abort the transaction) should be part of the database schema and not the application, since this strategy allows us to change the integrity rules without changing any application that uses the database. As a result of the separation of concerns, it is preferable to let updates of the database generate event occurrences that trigger integrity rule checks rather than letting the application check the integrity rules (as a part of the application).

Further, in active databases, different lines of reasoning (cf. Section 2.3 on p. 22) in the form of event contexts are useful. For example, in a weak integrity constraint that checks if the sum of salaries of all employees is reasonable, this sum of salaries can be evaluated once the transaction has been requested to commit. In this example, the recent event context is useful, since we are only interested in that something has happened (to a set of objects) rather than which objects were updated. To summarize, the separation of concerns makes event monitoring useful in active databases, an important class of type (i) applications. Additionally, event contexts can be employed to reduce the number of times integrity rules have to be checked.

In network monitoring, the reason for the lack of one-to-one relationships is different compared to the reason for active databases. In network monitoring, the reason is that there are several independent physical nodes (possibly co-operating) using the network. Each of these physical nodes may access the network independently. There is actually a state of the network, but the application can only view it as the sum of states. Further, states can be dynamically added and removed during operation, since physical nodes may be, for example, started and shut down.

Similarly to active databases, different lines of reasoning are also desirable for network monitoring. For example, chronicle event context is applicable for monitoring request/reply messages, whereas recent context is applicable to monitoring a boot process that may be restarted several times before the physical node is operational. To summarize, event composition is useful for network monitoring.

The reason why event composition is useful for application type (i) is the limited ability of finite state automata to model management of differ-
13.1 Evaluation of Event Composition in Real-Time Systems

ent lines of reasoning. Without loss of generality, this argument is applicable to any non-multithreaded program on any computer based on the von Neumann architecture. In conjunction with dynamic task creation and variables, it is possible to augment finite state automata to handle different lines of reasoning. However, this feature of managing different lines of reasoning is an integral part of event composition and can be specified in a less complex way than in finite state automata.

To be more precise, in finite state automata we can easily express linear time and cyclical time (of lines of reasoning); in conjunction with dynamic management of tasks executing the finite state automata, a parallel line of reasoning can be started by creating a new process. Also, branching time, where we only keep track of a single line of reasoning such as recent, is simple to model. However, the different forms of branching time that maintain several lines of reasoning simultaneously, represented by event contexts such as continuous and general, are not simple to model using finite state automata (even in conjunction with processes). In contrast to parallel lines of reasoning, in which the complete reasoning is contained within the state of one process executing a finite state automata, the branching lines of reasoning with multiple lines of reasoning result in that these lines of reasoning are spread out over several processes. This situation with lines of reasoning spread out over several processes is more difficult to control compared to the solutions where the line of reasoning is contained within one process.

![Finite state automaton diagram](image)

**Figure 13.3**: Finite state automaton for a sequence of $E_1$ followed by $E_2$

Concerning application type (ii): in a graphical user interface application, composite event type can be used for abstraction. For example, the finite state automaton in Fig. 13.3(a) can be compacted into the finite state
automaton in Fig. 13.3(b) on the previous page with similar semantics\(^3\). N.B., in contrast to event specifications, finite state automata specifications do not include different lines of reasoning. By using composite event specifications in finite state automata, it is possible to switch between different lines of reasoning with less effort; for example, switching from continuous event context to recent context requires a change of the composite event specification, whereas in pure finite state automata this change requires a transition from \(S_2\) to \(S_2\) to keep the most recent \(E_1\) occurrence.

To summarize, when there is not a one-to-one correspondence between the state of the application and the state of the controlled process, then event composition is useful for application type (i); in particular, in such situations that require continuous and general event contexts; these event contexts represent policies that have a left-branching view of time, that is, we need to keep track of several lines of reasoning at the same time. Further, even when event composition is unnecessary from an operational viewpoint (as for application type (i)), the abstraction provided by event composition can be shown to be useful for compacting system specifications (application type (ii)).

### 13.2 Formalization of Event Monitoring

Formalization of event monitoring is a challenging task, since event monitoring involves multiple aspects that need to be consistent. For example, the context predicates and the operator rules need to follow the requirements of our schema to be consistent, and to achieve this consistency may require a few iterations. In this Ph.D. thesis, the event specification language Solicitor is used as a case study to demonstrate the power of our formalized schema, for example, to prove the correctness of the definitions by the help of tables.

One important issue left out of this work is to ensure algebraic properties (e.g., associativity, distributivity) of the event operators. The reason for this is that the ability to prove algebraic properties depends on the event context. For example, in general event context, algebraic properties of event operators can be proven. However, in, for example, chronicle event context, several algebraic properties of event operators can be proven not to hold. Context

---

\(^3\)The exact semantics of an event expression equivalent to the finite state automaton in Fig. 13.3(a) on the previous page is defined by the event expression \(A(\text{context: continuous, overlapping: false})(\text{enter}S_1, E_1; E_2, \text{exit}S_1)\) assuming that \(\text{enter}S_1\) occurs when state \(S_1\) is entered and \(\text{exit}S_1\) occurs when state \(S_1\) is exited.
predicates that maintain algebraic properties can be expressed within our
schema, for example, the event context suggested by Carlson et al. [CL03]
(cf. Section 6.5.3 on p. 146).

For pragmatic reasons, it is desirable to allow context predicates that are
not required to ensure algebraic properties. Such a pragmatic reason is to
improve performance of event monitoring, since event contexts that do not
ensure algebraic properties can be less computationally expensive, in partic-
ular with respect to the size of the filtered event log, than their counterparts
ensuring these properties. For example, the context suggested by Carlson et
al. [CL03] is more computationally expensive for event composition based on
the sequence operator compared to event composition of any event operator
in recent event context (in, e.g., Solicitor). N.B., the other event operators
proposed by Carlson et al. follow the recent event context.

Another important issue in the formalization of event monitoring is that
tools for generating, understanding, and validating event types are required,
since event contexts make event expressions more difficult to understand
compared to, for example, predicate logic expressions. As mentioned earlier,
the reason for event contexts is to improve performance, but without appro-
priate tools it may be difficult to, for example, check that the result of event
composition is appropriate for the application.

13.2.1 Alternative Strategies to Formalization

The most notable formalization strategy of event composition is to start
with an unrestricted case (e.g., the general event context) and then add
filters for the various event contexts. This approach has been pursued by,
for example, Chakravarthy et al. [CM94] and Adaikkalavan et al. [Ada02,
AC02] and Carlson et al. [CL03]. The advantage of this strategy is that
any property proven in the general case also holds in the restricted case,
unless the property contradicts the filter (i.e., the constraint) in question.
Moreover, it can be more intuitive, since it requires less introduction as can
be seen in the work by, for example, Carlson et al. [CL03]. However, a
counterexample to the work by Carlson et al. is the work by Chakravarthy
et al. [CYY98].

However, the difficult part is to handle the chronicle event context, or,
to generalize, any event context only based on historical order. The recent,
continuous, and general event context all allow reuse of event occurrences of
at least one contributing event type (cf. Table 7.1 on p. 165). As mentioned
in Ch. 6, the problem is that predicate logic cannot handle consumption. One interesting approach would be to employ linear logic to define the event contexts, since it can handle consumption of truths. However, since linear logic or predicate logic need to be augmented to handle usage of event occurrences with respect to event types, where reuse may be allowed, we have chosen the more well-known predicate logic instead.

13.2.2 Forward and Backward Deduction

Another issue is how formalization affects event composition according to the event context (defining the policy in terms of lines of reasoning, cf. Section 4.5.4 on p. 55 and Section 2.3 on p. 22). Most approaches in event monitoring employ only forward deduction during event composition (e.g., all approaches stemming from active databases addressed in Ch. 4).

In forward deduction, new facts (i.e., composite event occurrences) are asserted using existing facts (i.e., contributing event occurrences). If we want to check that there is some combination of contributing event occurrences that meet, for example, a minimum duration in between them and only forward deduction is employed, then we need to generate all possibilities and apply a filter to remove incorrect lines of reasoning.

In contrast, both Chronicle recognition and monitoring of real-time logic use a hybrid of forward and backward deduction; when a new fact is asserted (an event has occurred), forward deduction is used to start up or branch off new lines of reasoning (composite event occurrences that may terminate in the future). The new facts are used in backward deduction to constrain what event occurrences can match the current incomplete lines of reasoning. For example, if there is a minimum duration between contributing event occurrences, then, as soon as a line of reasoning has been initiated, this line of reasoning can be associated with a time window in which forthcoming event occurrences are allowed to contribute if and only if their time span fits within this time window.

Hybrid solutions work better in the average case. Chronicle recognition (i.e., event composition in non-overlapping general context) is of \( O(\hat{g}\hat{e}^2) \) complexity in the average case [DGG93, Dou96]. Since chronicle recognition has been used for real-time systems by, for example, Milne et al. [MNG+94], this result is significant. In contrast, when forward deduction is used, as is the case in Solicitor, event composition is of \( O(\hat{g}^2) \) complexity for both average and worst cases (cf. Ch. 9). The key to successful usage of the
hybrid deduction is application of time constraints such as expiration time and minimum interarrival time in conjunction with a cyclical view of time (non-overlapping event context). This cyclical view of time means that when a composite event occurrence is completed, all other lines of reasoning that could result in other composite event occurrences are removed. N.B., if, for example, aperiodic events are allowed, then Chronicle recognition is of $O(\hat{g}^2)$ as discussed in Ch. 9.

13.3 Order and Time

Order and time are necessary to address on a system design level, since they have an impact on the rest of the system. For example, if it is possible to introduce a static resolution schema (based on quasi-order) among occurrences of related event types with termination times that cannot be discriminated (a requirement for Solicitor Section 6.3.1 on p. 120), then a computationally inexpensive algorithm can be used to partially order event occurrences. This relies on that stable requests [Sch94a] (in Section 4.6 on p. 63) are guaranteed.

The major problems with such a static resolution schema of timestamps are: (i) maintenance of event specifications, and (ii) managing conflicting requirements on the conflict resolution schema. Concerning case (i), when new requirements are introduced, the static ordering may have to be changed. In particular, in highly available systems the update of the event specifications (affecting the static resolution schema) must be performed on an operational (sub)system. It is possible to use information about expiration to introduce an update of the event specifications incrementally. For example, when an event type is updated, the old and new event types co-exist in the system for the expiration time of the old event type. The subscribers receive composite event occurrences of the old event type from the time of the update of the event specifications during the expiration time; after expiration, the subscribers start receiving event occurrences from the new event type and the old event type can be removed. Given the strategy in this example, we ensure that the state of event composition is correctly transferred from the old event type to the new event type.

Concerning case (ii), our proposed conflict resolution schema is based on a global ordering of event types that contributes to the same composite event type. If there are common subexpressions, then there may be conflicts
in this ordering. A simple way is to avoid common subexpressions or to reach an acceptable compromise of global ordering.

The alternatives to a static order resolution that prevent concurrent event occurrences are, as discussed in Section 4.6 on p. 63, either to avoid or allow concurrent event occurrences. Briefly, avoidance requires centralized control and allowance can end up in computationally expensive solutions such as the approach suggested by Schwidersky et al. [Sch95b].

13.4 Performance and Efficiency Considerations

There are four major issues for performance (and efficiency) in event composition:

1. indexing of event types

2. invalidation of partially instantiated composite event occurrences (for connector set management and memory management) and management of duplication

3. intra-task and inter-task communication of event occurrences

4. how to design event composition with respect to the results of the time complexity study

**Issue 1:** Indexing of event types is important, since event composition relies on placing event occurrences in the correct connector sets. Naïve solutions, such as using no index, degrade the performance significantly as was demonstrated in the BEAST benchmark [GBLR98]. In the implementation of Solicitor, a standard C++ map template (e.g., [LL98]) was used to index the event types. Standard C++ map templates are balanced trees. If the number of event types is large, then hashing can be used to further improve the lookup. If event monitoring is divided into several tasks, then indexing of event types becomes even more pronounced, since the lookup must be performed in each event monitoring task.

**Issue 2:** A critical issue is how invalidation can be optimized in such a way that only one operation is required to perform invalidation of several initiated, but not terminated, composite event occurrences. There are two
significant situations (cf. Sections 10.6.2 to 10.6.3 on p. 235): (i) in a system with a sparse time base, several expirations are grouped with the same timeout, and (ii) when duplication is employed (cf. Section 4.5.6 on p. 62).

Concerning situation (i), Liu et al. [LM99] report that only a few percent of initiated but not terminated composite event occurrences terminate in network monitoring. This means that most composite event occurrences do expire. Basically, if both representations of event occurrences and elements in the connector sets pointing to representations of event occurrences are stored in memory allocated from memory pools indexed on expiration time points\(^4\), then it would be possible to deallocate all memory for event occurrences as well as elements in the connector sets with one operation. The only other issue is that the connector sets must be aware that elements can be deallocated and should be ignored when iterating over the connector set to select event occurrences that match the event context. This optimization has not been pursued in this Ph.D. thesis.

Concerning situation (ii), in some event contexts such as general and continuous, duplication is allowed (cf. Section 4.5.6 on p. 62 as well as Section 10.12.8 on p. 244). Duplication implies that there are composite event occurrences that share contributing event occurrences as a result of consecutive results from the same composite event type. In conjunction with invalidation (cf. Section 10.6 on p. 234), this is an important issue. The reason is that it is desirable to invalidate several initiated event occurrences at the same time. For example, Dousson et al. [DGG93, Dou96] advocate that once a chronicle (i.e., a composite event occurrence in chronicle recognition) has been completely instantiated, then all other incompletely instantiated chronicles should be removed. Therefore, management of duplicates is of major interest for efficient implementation of event composition. This duplication issue of invalidation has not been addressed within this Ph.D. thesis.

**Issue 3:** Intra-task and inter-task signalling of event occurrences can be a performance bottleneck. By only copying memory when it is necessary by, for example, using safe pointers [Mel00], it is possible to reduce intra-task signalling. Also, by aggregating event occurrences into messages based on destination task, it is possible to reduce the number of messages sent between tasks and thereby reduce the computational cost of inter-task signalling.

\(^4\)In a system with a spare time base, a time point represents a duration of notable size.
**Issue 4:** The single most important variable to limit is $\hat{g}$ (i.e., the size of the filtered event log), even though the time complexity of event composition in both recent and chronicle context is independent of this variable. The reason is that the size of this input is likely to be several magnitudes larger than any other parameter; a fact that can have a great impact on event composition in both continuous and general event contexts, since event composition in continuous context is of linear time complexity with respect to $\hat{g}$ and event composition in general context is of polynomial complexity with respect to $\hat{g}$. Even though the chronicle event context is independent in terms of time complexity, it is of linear space complexity, since every event occurrence is saved until it is consumed.

The next most significant issue for recent, chronicle, and continuous event contexts, is the use of safe pointers, since usage of safe pointers reduces the time complexity from $O(\hat{e}^2)$ to $O(\hat{e})$ in all but the general event context.

The next most significant issue for general event context is to limit the event type size, since event composition in the general event context is of exponential time complexity. This factor is not considered the most significant issue for general event context, since, as mentioned in Section 12.3.1 on p. 274, event expressions are expected to be small due to the difficulty of understanding event composition [BMH99].

The least significant parameter from a time complexity perspective is the parameter size. In recent, chronicle and continuous event contexts, event composition cost scales linearly with respect to this parameter.

### 13.5 Support for Real-Time Systems

In addition to the timeliness of event composition, event monitoring must meet the requirements of the infrastructure of a real-time system. This infrastructure contains means for achieving timeliness, for example, a real-time application is split into tasks to allow real-time scheduling of these tasks.

By explicitly dividing the monitor specification into tasks that group composition for a set of event types, it is possible to enable a real-time scheduler to meet time constraints. For example, each task may have different time constraints as well as criticalities. Thereby, this architecture meets the requirements of event criticality addressed by Berndtsson et al. [BH95] (cf. Section 4.10 on p. 85). The grouping of event specifications into
tasks with the given criticality is an extension of their idea. Thereby, event criticality can be mapped to criticality of tasks. Moreover, partitioning a monitor specification into tasks is a sound basis for, for example, allocating these event monitoring tasks to different processors.

The major disadvantages of explicit grouping of event specifications into tasks are: (i) it can be more difficult to update the monitoring specification in such a system, since a change may imply that an event type should be monitored by a different monitoring task, and (ii) it may inhibit reuse of monitor specifications. For example, one monitor specification can use an event specification of another monitor specification, but the grouping into tasks may make this reuse impossible or require intertask signalling of event occurrences.

13.6 Application Examples

This section covers examples of major application areas of event monitoring such as fault management and active databases.

13.6.1 Fault Management

Fault management in critical applications can be supported, since relative temporal events can be specified and expiration can be used to end a line of reasoning. There are two parts of fault management that are of interest as discussed in Section 3.3 on p. 32: fault tolerance and fault removal.

Fault Tolerance

We start with a simple example without parameterized event types to explain the specification. This example is followed by a discussion concerning parameterized event types.

Example 13.1

Network monitoring: Omission failures in Remote procedure calls:
Assume that remote procedure calls are monitored, where call and reply are primitive event types. The event type call is signalled when a remote procedure call is initiated and reply is signalled when a reply has been received. Assume that there is at most one process issuing remote procedure calls and let the following event specification be monitored in chronicle event context:
• \textit{callFailure} = N(call, reply, call+T) \\
occurs if a call is initiated, but no reply is received within \(T\) duration \\
• \textit{callSuccess} = N(call, call+T, reply) \\
occurs if a call is initiated and a reply is received within \(T\) duration \\
• \textit{callLateSuccess} = callFailure;\langle\text{expire}:10T\rangle reply \\
occurs if a call is initiated and a reply is received within \(10T\) duration \\

Where \(T\) is a duration and \(c_{\text{ord}}(\text{call}) < c_{\text{ord}}(call+T) < c_{\text{ord}}(\text{reply})\) according to Lemma 8.5 on p. 187.

This example would not work with termination semantics, since the relative temporal event occurrence of overlapping call/reply sequences may inhibit incorrectly. However, since interval semantics is used, it will work since two call event occurrences can receive discriminating timestamps.

However, it is necessary to use parameterized event types for a real situation with multiple processes performing remote procedure calls. For example, \textit{callFailure(int pid)} = N(ptype: pid)(call, reply, call+T)\(^5\) where \(ptype: \text{pid}\) indicates that the \(\text{pid}\) parameter is used for parameterization. That is, this specification means that there is a \textit{callFailure\_pid} specification for each \(\text{pid}\). The reason is that such a monitor can be global. Parameterized event types are considered future work Section 15.3.13 on p. 317.

One problem with failures is redundant event occurrences of the same type. For example, if diffusion is used to send \(k\) call messages, then there are up to \(k\) replies. Any number of replies larger than one is a success. The rest of the replies may be thrown away. There are a number of solutions to this:

• use either non-overlapping general event context or non-overlapping recent event context instead of chronicle event context, this leads to that when one composite event occurrence (a pair of call and reply events has occurred) is generated, then all other possible solutions are thrown away since they overlap with the first composite event occurrence. N.B. if a diffusion protocol is used, then the next request

\(^5\)Alternatively, \textit{callFailure(ptype}: \text{pid})(\text{int} \ \text{pid}) = N(\text{call}, \text{reply}, \text{call+T}).
cannot be sent before we expect the last reply. The reason for this enforced delay is that replies to the new request may be mixed up with replies to the old request.

- assign a unique request identity to each call and parameterize the specification with respect to the request identity.

Fault Removal

Event monitoring can, as mentioned in Section 2.1.3 on p. 15 and Section 3.3 on p. 33, be used to provide observability (cf. [Mel98b]). In particular, if event specifications are used for on-line usage of the system (monitored object) itself, then these specifications can be used to test the applications; it is possible to use the event specifications to generate test input as well as to analyze the test results.

For example, in Example 13.1 on p. 291, these event specifications can be used to generate all permutations of temporal ordering of call events and reply events. It is conjectured that it is possible to analyze the event specifications to obtain interesting timestamps, for example, generate three test cases for a given event specification: one test case that completes within the expiration time, one test case that is as long as the expiration duration, and one test case that fails to occur within the expiration time.

13.6.2 Active Databases

Event monitoring is the core support for active (real-time) databases (cf. Section 3.4 on p. 33). Basically, it enables the use of condition-action rules (CA-rules) in the form of event-condition-action rules (ECA-rules) in databases by significantly reducing the number of times the condition-part of the rules has to be evaluated. Without event monitoring, the rules must either be evaluated (i) after each database operation, or (ii) periodically. Case (i) can lead to significant performance degradation. Case (ii) leads to less performance degradation compared to case (i), but is hard to control in real-time systems, since there is no connection between an update to the database and when the rules are evaluated.

From a database perspective, the main issue in Solicitor is to provide this core service with predictable resource requirements so that event composition can be used in real-time databases. Further, Solicitor provides the basis for a solution to the problem of monitoring outside the scope of transactions.
by adding expiration time to the event specifications. Given temporal (real-time) databases, that is, databases containing temporal data, the expiration time can be used to expire initiated event compositions. For example, the relative and absolute temporal validity intervals of data [Ram93] can be used to derive expiration times (and vice versa).

Example 13.2

**integrity constraints example:**
Assume that there are three scalar variables in the database: \( x, v_1 \) and \( v_2 \). The integrity constraint is that \((v_1 < x \land v_2 = 2x) \lor (v_1 \geq x \land v_2 = 3x)\). When a variable is updated, an \( update_k \) is generated for variable \( v_k \). Given this, then the ECA-rules “checkIntegrityA” and “checkIntegrityB” maintain the integrity constraint:

**Rule** checkIntegrityA:
ON: \( update_1 \lor (context: recent)update_2 \)
DEFERRED IF: \( v_1 < x \land v_2 \neq 2x \)
DO: ABORT TRANSACTION

**Rule** checkIntegrityB:
ON: \( update_1 \lor (context: recent)update_2 \)
DEFERRED IF: \( v_1 \geq x \land v_2 \neq 3x \)
DO: ABORT TRANSACTION

These ECA-rules have their condition evaluation deferred to the end of the transaction. The event type follows the recent context, since we do not need to identify the source of the event occurrence to check the constraint. When an update of \( v_1 \) or \( v_2 \) occurs, then both ECA-rules are triggered.

Instead of aborting the transaction, we can also let the rules maintain integrity constraints. In the ‘maintainIntegrity’, which is triggered by the update of \( v_1 \), the \( v_2 \) variable is updated at the end of the transaction.

**Rule** maintainIntegrity:
ON: \( update_1 \)
DEFERRED IF: \( true \)
DO:

\[
\text{if } v_1 < x \text{ then } v_2 := 2x; \\
\text{else } v_2 := 3x;
\]

The ‘maintainIntegrity’ rule is not triggered if \( v_2 \) is updated. This requires an additional ECA-rule that conflicts with the ‘maintainIntegrity’ rule. However, we can mix all three suggested rules, but it is necessary to
prioritize the evaluation of these rules. For example, we need to evaluate the 'maintainIntegrity' rule first followed by the different checks.

Example 13.3
Monitoring of external events:
In the licentiate thesis [Mel98a], several examples covered monitoring of external events. For example, Fig. 13.4 depicts a situation where a vehicle enters from the left and exits via one of the two routes. Assume that it takes $T_1$ time to pass from sensor $S3$ to either sensor $S411$ or $S421$. The database application should maintain a variable $route$ whose value should either be 1 when passing sensor $S411$, or 2 when passing sensor $S421$, after sensor $S3$ has been passed. Until the vehicle passes either $S411$ or $S421$, the value of $route$ should be 0.

$$
\text{vehicles exits}
\begin{array}{c}
S421 & S422 \\
S3 & S411 & S412
\end{array}
$$

Figure 13.4: Alternative route example [Mel98a]

The important issue is to ensure that initiated event occurrences are terminated irrespective of which alternative is chosen. There are three principal ways of specifying this (in descending order of efficiency): case (i) let event monitoring keep track of which alternative that was chosen, case (ii) let the condition evaluation sort out which alternative that was chosen, and case (iii) let the action part sort out which alternative was chosen.

In case (i), “route1a” and “route1b” updates the $route$ variable correctly. The occurrence of one alternative stops event composition of the other alternative. This is the most efficient way, since condition evaluation is avoided as long as possible and only one rule is triggered for evaluation.
Rule route1a:
ON: \( N(S1, S421, S411) \)
IF: \( \text{true} \)
DO: \( \text{route} := 1 \)

Rule route2a:
ON: \( N(S1, S411, S421) \)
IF: \( \text{true} \)
DO: \( \text{route} := 2 \)

In case (ii), “route2a” and “route2b” perform the same thing as in case (i), but in this case the evaluation of which alternative that has actually occurred is left for the condition evaluation. Condition evaluation is triggered for both rules. This is less efficient than case (i).

Rule route1b:
ON: \( E = S1:(S411\lor S421) \)
IF: \( \text{type(terminator}_E = S411 \)
DO: \( \text{route} := 1 \)

Rule route2b:
ON: \( E = S1:(S411\lor S421) \)
IF: \( \text{type(terminator}_E = S421 \)
DO: \( \text{route} := 2 \)

Finally, in case (iii), the “routes” rule maintains the route variable. This is the least efficient way, since the check is not performed until the action is executed. It checks the condition in the action execution, but only one rule is triggered for evaluation. This case is approximately as efficient as case (ii). However, in terms of analyzing the ECA-rules, case (ii) is better than case (iii), since conditions are easier to analyze compared to arbitrary code.

Rule routes:
ON: \( E = S1:(S411\lor S421) \)
IF: \( \text{true} \)
DO:
\[
\begin{cases}
\text{if } \text{type(terminator}_E = S411 \\
\text{then } \text{route} := 1; \\
\text{else } \text{route} := 2;
\end{cases}
\]
Chapter 14

Related Work

“One of the symptoms of an approaching nervous breakdown is the belief that one’s work is terribly important.”

/Bertrand Russell

This chapter addresses other existing approaches to various aspect of event monitoring, using the themes addressed in the chapters of this Ph.D. thesis: Formalization of event composition (Ch. 6), realization from a formal specification (Ch. 7), limitation of processing (Ch. 8), algorithmic time complexity of event composition (Ch. 9), architecture of event monitoring (Ch. 10), and memory management for event composition (Ch. 11). Each of these chapters contain a detailed related work section, whereas this chapter addresses certain overall aspects.

14.1 Formalization of Event Composition

A brief comparison of other approaches follows (a more detailed comparison of some of the approaches are found in Section 6.5 on p. 142). Except where otherwise noted, all other approaches presented in the rest of this section are based on termination semantics rather than interval-based semantics.
Snoop: 1st attempt

To summarize the comparison in Section 6.5.1 on p. 143 between our work and the work by Chakravarthy et al. [CYY98], in contrast to our work, they do not explicitly address scheduling of event composition. They define the operators in the general event context for event histories. Based on these operator definitions, filters for each event context can be applied. In contrast to our work, there is no clear separation between used and invalidated event occurrences; a fact that makes their approach more complicated, since they need to specify this indirectly in their filters.

Snoop: 2nd attempt

To summarize the comparison in Section 6.5.2 on p. 145 between our work and the work by Adaikkalavan [Ada02] and Adaikkalavan and Chakravarthy [AC02], this attempt is an extension to Galtons [GA01] work. They take the predicates for each event operator and add context specific parts. In contrast to our approach, they cannot handle consuming event contexts such as the chronicle event context. Moreover, in contrast to our work, their approach does not clearly separate specification of event operators and event contexts. Similarly to our approach, interval-based semantics is supported.

SAMOS

The approach defined by Gatziu [Gat94] is semi-formal, since she relies on other work to define Petri-nets. Only one event context (chronicle) is supported and, therefore, there is no need to distinguish between event operator definitions and event context definitions. Scheduling of event composition is not separated from the definition of their event specification language.

ODE

Gehani, Jagadish et al. (e.g., [GJ92b, GJ92a, JMS92, GJS93, GJM93]) have a formally defined event specification language based on finite state machines. In contrast to our work, they do not separate event context definitions from the event operator definitions, since the context definition relies on the event operator definition [ZU99]. Moreover, they do not address scheduling of event composition.
14.1 Formalization of Event Composition

Monitoring of Real-time Logic

Monitoring of Real-time Logic has been addressed by several sources by, for example, Mok, Jahanian, Liu etc. [JM86, CJD91, ML97a, ML97b, LMK98]. In contrast to Solicitor, real-time logic has fewer event operators (cf. Table 4.3 on p. 46). However, there is no clear distinction between event expressions and the event contexts. Instead, the event contexts are integrated into the event composition mechanism. There is no need to address scheduling of event composition explicitly, since their only event operator does not introduce any arbitrary decisions.

Temporal Calculi For Communicating Systems

In contrast to our approach, Bækgaard et al. [BG97] do not address either scheduling of event composition explicitly or event context definitions. The reason for the latter is that they only support the chronicle event context. One interesting feature they address that is lacking in our approach, is parameterized event types (cf. Section 4.8.2 on p. 67). This is future work (in Section 15.3.13 on p. 317).

Chronicle Recognition

The formalism of Chronicle recognition by Dousson et al. [DGG93, Dou96] is based on temporal constraint satisfaction programming (cf. Section 4.8.2 on p. 67). Therefore, it has a sound mathematical background. In contrast to our approach, Dousson et al. do not separate event operator definitions and event context definitions, since they only support the (non-overlapping) general event context. Moreover, they do not address scheduling of event composition explicitly even though they have the disjunction operator. The reason is that they are satisfied to find one sequence of event occurrences that meet the time constraints rather than all.

Event Pattern Language

The formalism of EPL by Motakis et al. [MZ97] is based on DataLog\_IS and is machine-executable (cf. Section 4.8.2 on p. 67). Therefore, their formalism is closer to a real production system than our formalized schema. However, their formalism is more tightly coupled to transactions than Solicitor. Similarly, they can express event specification languages in EPL; however, since
their approach is closely connected to transactions, they do not clearly stress
the similarities and dissimilarities between different context predicates as is
done in our work (e.g., the conclusion summarized in Table 7.1 on p. 165 for
the context predicates in Def:s 6.39 to 6.42 on pp. 129–130 are not explicitly
visible in their approach). Scheduling of event composition is not addressed.

14.2 Realization Issues from a Formal
Specification

A brief comparison in terms of realization follows (cf. Section 7.4 on p. 174
for a more detailed comparison). Only our schema allows realization from
all event contexts.

Snoop

In contrast to our work, there are no attempts to provide a method for
using the formalism for realizing an event monitor in the Snoop attempts.
Moreover, scheduling of event composition has not been addressed.

SAMOS

Gatziu [Gat94] only reference standard implementation of Petri-nets.

ODE

The work of Liuwen et al. [LGA96] addresses a number of issues with re-
spect to the implementation of event composition, but none address how to
translate their formalism into an implementation.

Monitoring of Real-time Logic

The underlying theory, which is based on graphs, has well-defined algo-
rithms that unambiguously define how event expressions are processed (in
recent and chronicle event contexts). These graphs contain data about event
types and their relation in terms of time constraints to other event types.
Architectural issues are addressed by Liu and Mok [LM99].
Temporal Calculi For Communicating Systems

Neither Bækgaard et al. [BG97] nor Miller [Mil89] address how specifications are translated into an implementation.

Chronicle Recognition

Chronicle recognition is based on temporal constraint satisfaction programming and, hence, there are precise definitions for realization for their combination of overlapping and non-overlapping general event context.

Event Pattern Language

EPL [MZ97] is machine-executable and is therefore closer to the machine than Solicitor. The disadvantage of their approach is that proof-techniques for predicate logic are not immediately applicable compared to Solicitor.

Deriving Real-time Monitors

Peters [Pet00] addresses the issue of how to derive real-time monitors from a formal requirements specification. Peters has addressed a lot of important issues with respect to monitoring of real-time systems with respect to the requirements. It is conjectured that his approach could generate monitor specifications for Solicitor. However, this remains to be demonstrated (and is addressed as future work in Section 15.3.10 on p. 316).

Summary of Realization Comparison

To summarize, our approach is alone in providing a method for translating context predicates into invariants useful for implementing an event composer as well as precisely and unambiguously defining correct scheduling of evaluation of event types, while the underlying formalized schema retains the possibility of applying proof techniques for first-order predicate logic.

14.3 Bounded Resource Requirements of Event Composition

The only source that actually presents a proof is the work of Mok et al. [ML97a]. Essentially, the main advantage of our proof is that it is more
general than the proof presented by Mok et al. [ML97a] with respect to event contexts as well as actual implementations of event monitoring. The main disadvantage of our proof is that it is not based on an actual algorithm that can be immediately translated into efficient object code, whereas the proof by Mok et al. [ML97a] is based on algorithms that can be immediately translated into efficient machine-executable code. For more information, please read Section 8.5 on p. 193.

14.4 Algorithmic Time Complexity of Event Composition

None of the other approaches addressed in this work (cf. Section 9.6 on p. 215 for a more detailed comparison) address algorithmic time complexity of the complete event monitoring processing. Chronicle recognition [DGG93, Dou96] and monitoring of real-time logic (e.g., Mok et al. [ML97a, ML97b, LMK98]), which are the only approaches addressing time complexity, address only the algorithmic time complexity of the matching and generation of composite event occurrences for the single terminator case. The results by Mok et al. are proportional to our results.

In contrast to our work, the work in Chronicle recognition addresses the average case, whereas we are addressing the upper bound time complexity. An important point in favor of Chronicle recognition is that with appropriate constraints in the monitored object (e.g., monitoring the environment in a time-triggered real-time system), their upper bound time complexity is equivalent to their average case time complexity. This average case time complexity is significantly less than the upper bound time complexity as a result of that chronicle recognition employs a hybrid between forward and backward deduction (cf. Ch. 6 and Section 13.2.2 on p. 286). We conjecture that the necessary constraints to achieve this equivalence of worst-case and average case complexity are as follows: appropriate time constraints must be set, sporadic events must be supported, and a cyclical view of time must be used (i.e., non-overlapping event contexts). However, in applications with inappropriate constraints (e.g., no time constraints at all), the upper bound time complexity is exponential.

In active databases, there has been work on space complexity of the internal representation (e.g., event graphs, finite state automata) of event types by, for example, Deutsh [Deu94], but none has addressed the algorithmic time complexity of event monitoring.
14.5 Architecture of Event Monitoring

A detailed comparison of architecture is discussed in Section 10.12 on p. 238. The sole purpose of the proposed architecture of Solicitor is to enable monitoring of events, with predictable resource requirements, for embedded real-time systems. In contrast, the purpose of event monitoring in active databases is to optimize performance of the rule-processing (e.g., [DBB⁺88, CBB⁺89]). Therefore, many approaches integrate event monitoring in rule processing (e.g., [BZW95]). In comparison to Snoop [CM94, CKAK94], GEM [MSS97], COBEA [MB98], SMILE [Jae97], X²TS [LCB99], and JEM [LM99], Solicitor is unbundled from the rest of the system. In contrast to, for example, COBEA, and X²TS, Solicitor does not provide interoperability for a heterogenous distributed system. Similarly to Chronicle recognition [DGG93] and monitoring of real-time logic [ML97a, ML97b], expiration management is addressed by using timers. In contrast to the work of Schwidersky [Sch96], expiration is not initiated at an event type and then propagated to the contributing event types. Rather, each event type inherits the expiration time and sets timers when initiator occurrences arrive.

Support for transactions is inherent in active database prototypes such as SAMOS [Gat94], REACH [BZW95], and ODE [GJ92b, GJ92a, LGA96]. An advantage of transactions is that event composition can be terminated when a transaction terminates. However, as discussed by Buchmann et al. [BZW95] it is desirable to monitor composite event types where the contributing event types stem from different transactions.

In most approaches, event monitoring is implicitly invoked, event composition scheduling is data-driven, and the result is immediately delivered to the subscribe; SMILE [Jae97], and GEM [MSS97] are exceptions. However, no other approach has the same configurability as Solicitor.

Concerning the software organization of the event monitor, Solicitor is similar to, for example, SMILE [Jae97] and COBEA [MB98], in that there are functionalities for collecting event occurrences, performing event composition, and delivering the result. However, Solicitor is the only architecture that addresses how this functionality is mapped to tasks that are schedulable units of a real-time system. Memory management for event composition, one of the key contributions of this Ph.D. thesis, has not been addressed in any other architecture.
Chapter 15

Conclusions

“And now, the end is near, and so I face, the final curtain.
My friend, I’ll say it clear,
I’ll state my case, of which I’m certain.
I’ve lived, a life that’s full, I’ve traveled each and every highway.
And more, much more than this,
I did it my way.”

/Paul Anka and Frank Sinatra

15.1 Summary

Event composition is useful for real-time systems and, in particular, required for active real-time database systems. It is a powerful feature for real-time systems, since event composition can handle aggregation and filtering of event occurrences, an aspect that is important in real-time systems. In particular, event-triggered real-time systems are prone to event showers, where aggregation and filtering are the major means of handling event showers.

A requirement of tasks (executing algorithms) in real-time systems is that they are timely. Therefore, it is necessary to demonstrate that the algorithm performing event composition is timely. That is, the algorithm performing event composition must have predictable resource requirements as well as be sufficiently efficient. To prove that an algorithm is predictable, a formal model is necessary.
Unfortunately, existing formal models are either limited to particular event contexts (that control the event composition processing) such as monitoring of real-time logic and temporal calculi for communicating systems, or do not consider processing explicitly (e.g., formalizations of Snoop [CYY98, Ada02, AC02]). Therefore, a formalized schema is required to define event specification languages.

A formalized schema is presented and as a proof of concept, the event specification language Solicitor (that is derived from Snoop), has been specified. The strengths of the formalized schema have been demonstrated by deriving significant invariants for event composition as well as defining correct scheduling of evaluation of event types in an event specification. We have demonstrated that event composition has predictable resource requirements; that is, the number of steps required to perform event composition given that each event type has an associated (specified or inherited) expiration time and that each contributing primitive event type is sporadic. This proof holds for all event operators in all event contexts. Moreover, the time complexity of event composition in different event contexts has been derived for Solicitor. The accuracy of the computational model that is the base for the investigation of time complexity has been studied by experiments using an implementation of the proposed architecture for event monitoring.

An architecture for event composition meeting real-time requirements has been proposed. In particular, it allows separation of event monitoring into tasks, where each task may have different time constraint specifications, criticality etc. Also, this architecture clearly separates implicit and explicit invocation of event composition as well as immediate and delayed delivery; this feature allows an application designer to choose an appropriate configuration of event monitoring, for example, if implicit invocation is sufficient, then this removes the requirement of explicitly initiating event composition and, thereby, reducing the complexity of the application.

In this architecture, critical parts that significantly affect the resource requirements of event composition have been identified. These critical parts of the architecture are scheduling of evaluation of event types as well as expiration management. These parts can use bit-manipulating data structures to improve the efficiency of the underlying algorithms of these services.

We have introduced a predictable and efficient solution to the memory management bottleneck in event composition. This memory management handle event parameters (defined by both the system and the user). This solution can make event composition useful for a broader range of appli-
cations. The main contribution is the idea of combining self-referencing memory and safe pointers, which can significantly improve the performance of composite event monitoring (we have shown that event composition based on safe pointers scales better than event composition based on unsafe pointers as well as we have obtained that there is up to 20 percent less overhead for event types of size 1). This solution is bounded given that the event specifications do not introduce any cycles and that time constraints are applied to bound lengths of internal queues (i.e., expiration time and minimum interarrival time constraints are applied). The solution is also optimized for cloning, which is required when we want to signal an event to a subscriber (in a different address range). An advantage of self-referencing memory is that the computational cost of signaling several subscribers is reduced due to that the first clone can be used as a template to create other clones for other subscribers. We also make use of inlining (i.e., function definitions are lexically substituted into the code where they are called in a similar way as a macro), which maintains the abstraction of a function call while offering better performance (for small-sized functions).

15.2 Contributions

The major contributions of this Ph.D. thesis follow the layout of Part II and Part III, where the contributions from each chapter are classified as belonging to problem definition, method, or results.

Briefly, the following list states the main contributions (grouped according to the major themes) of this Ph.D. thesis:

1. Formalized schema for event composition
   
   (a) The need for such a schema has been identified.
   
   (b) A formalized schema for event composition has been defined.
   
   (c) We have clearly separated scheduling of evaluation of event types, event context semantics (i.e., selection of candidate event occurrences), and the event operator semantics (i.e., the evaluation if the candidate event occurrences can form a composite event occurrence)
   
   (d) The formalized schema prescribes properties (based on axioms) to enable correctness proofs of event operator specifications.
(e) The important concept of earliest available event occurrences has been discovered and has been extensively used in our schema.

2. Proof of concept: The Solicitor event specification language

(a) Solicitor has been defined using the formalized schema.
(b) Invariants for the connector sets have been derived from the specification of Solicitor.
(c) We have defined a scheduling algorithm of event composition that yields correct results regardless of the event specification language; this algorithm is based on the abstract computational model in the formalized schema.
(d) We have shown that the event contexts proposed in Snoop are the complete set of basic event contexts.

3. Bounded event composition

(a) Event composition has been proved to have predictable resource requirements, that is, the resource-predictability hypothesis holds (cf. Ch. 5).
(b) We provide the precise and unambiguous definition of early detection of expiration of any event expression in any event context in any event specification language that is possible to define in our formalized schema.

4. Algorithmic time complexity

(a) Algorithmic time complexity of event composition has been investigated.
(b) The scalability hypothesis (cf. Ch. 5) that the time complexity of event composition is polynomial or better for all variables has been shown to be true, except for the dependency of event composition in the general event context on \( \hat{e} \).
(c) The results of the investigation of time complexity have been corroborated by experiments.

5. Architectural issues

(a) Features that improve efficiency and provide necessary ability to configure event monitoring have been identified and solved.
15.2 Contributions

(b) Memory management for event composition based on a novel combination of self-referencing memory and safe pointers has been developed and investigated.

15.2.1 Formalized Schema for Event Composition

Contribution 1a: The need for a formalized schema for event composition has been identified: a formalized schema is necessary since existing formalisms do not support event contexts properly in terms of proving properties about the processing. Either these formalisms are limited to specific contexts (e.g., TCCS is limited to the chronicle event context) or they include all contexts without paying attention to the processing (e.g., formalisms of Snoop).

Contribution 1b: A formalized schema for event composition has been defined in this Ph.D. thesis. The major feature (Contribution 1c) of this schema is that it separates scheduling of event composition, event operator semantics, and event context semantics. As a result of this separation, we are able to reason about, for example, correct scheduling of event composition as well as define event operator rules that are generic with respect to event context predicates and vice versa.

Contribution 1d: The schema also prescribes properties of event operator rules such that it is possible to verify the program safety (i.e., nothing bad will ever happen) of the rules by separating different relevant cases and demonstrating that there is only one applicable operator rule for each case.

Contribution 1e: These proofs of program safety are based on the precise definition of earliest available event occurrences, a central concept that simplifies the understanding of event composition. It simplifies the understanding, since it is a fix point that can be used to address what happens during event composition.

15.2.2 Proof of Concept: Solicitor

Contribution 2a: Our formalized schema has been used to define the semantics of Solicitor. In this event specification language, a simple time model is used since we advocate computationally inexpensive quasi-order enforcement of simultaneously terminating event occurrences of event types contributing to the same composite event type. In other words, all event occurrences of event types contributing to the same event type are partially ordered if they terminate simultaneously. This order is introduced by a (static) ordering of the event types to resolve such conflicts. Since this ordering is static, it
causes less overhead compared to other timestamping policies such as the method suggested by Schwidersky [Sch95b]. Additionally, if stable requests (messaging) are guaranteed, then the conflict resolution of simultaneously terminating event occurrences of related event types can be decentralized.

The strength of our formalized schema and Solicitor has been demonstrated by deriving important invariants (Contribution 2b) as well as defining how event composition should be scheduled to achieve correct results for any event expression (Contribution 2c). These invariants can be used to prove or generate algorithms that perform event composition according to the operator rules and event context predicates. These invariants use the context predicates. We have provided a proof outline of the pseudocode for the sequence operator in chronicle event context to demonstrate the ability to prove correctness of event composition by using the derived invariant. The specification of Solicitor also provides important auxiliary variables to help perform proofs.

A side effect of the derivation of invariants is the conclusion that the proposed event contexts from Snoop are the complete basic set in terms of historical and relative order-based constraints (Contribution 2d).

15.2.3 Bounded Event Composition

Contribution 3a: Given that all event types are associated with an expiration time (either explicitly specified or implicitly derived), then event composition has been proved to have bounded resource requirements.

Briefly, the proof has been done in the following way: given that each event type has an associated expiration time and that each primitive event type has a minimum interarrival time, then we are able to prove the equivalence of an expression based on relative temporal event operators with the operator rules stating that the span of an event occurrence should be less than or equal to the expiration time. This proof holds for any event context and for any event operator that can be defined within our formalized schema.

The significance of the proof is that expiration must be enforced by using timeouts; timeouts allow us to enforce bounded filtered event log. If timeouts are not used, then the filtered event log may grow indefinitely since expiration is not checked until a terminator occurrence is signalled (if ever). The precise and unambiguous semantics of expiration of an event type (Contribution 3b) is based on an equivalent expression whose principal operator is chronicle aperiodic.
15.2 Contributions

15.2.4 Algorithmic Time Complexity of Event Composition

Contribution 4a: The algorithmic time complexity of event composition has been studied. The computational model is the specification of Solicitor in our formalized schema with necessary adaptations (e.g., constraints, transformations) to ensure that this computational model is a sufficiently accurate definition of an implementation of event monitoring. For example, a necessary adaptation is how the filtered event log is implemented, a fact that is not covered in the schema. Another example of an adaptation is whether safe or unsafe pointers are used in the implementation of event monitoring.

This study concluded that the major variables affecting event compositions are the size of the filtered event log (\(\hat{g}\)) and the size of an event expression (\(\hat{e}\)). Further, the scalability hypothesis holds except for the dependency of event composition in general event context on \(\hat{e}\) (Contribution 4b). A more detailed summary follows.

In recent and chronicle event context, the size of the filtered event log has no impact (i.e., it is of \(O(1)\) time complexity with respect to \(\hat{g}\)) in contrast to the continuous and general event context (i.e., they are of linear, \(O(\hat{g})\), and polynomial, \(O(\hat{g}\hat{e})\), time complexity respectively). In general event context, the size of event expression has more impact on event composition than the size of the filtered event log, since the time complexity is exponential with respect to \(\hat{e}\).

An important part of the study is that safe pointers reduce the time complexity with respect to \(\hat{e}\) from polynomial (\(O(\hat{e}^2)\)) for unsafe pointers to linear (\(O(\hat{e})\)) for safe pointers.

The accuracy of the complexity class has been validated in an empirical study. This empirical study corroborates (Contribution 4c) the theoretical results.

15.2.5 Architectural Issues Concerning Event Monitoring

Contribution 5a: In the architecture, the following contributions are made: (i) explicit grouping of event specification into tasks, (ii) a clear separation between invocation of event composition, scheduling of event composition, and delivery of event occurrences, (iii) identification of parts that can be bottlenecks (i.e., the scheduling of event composition, the timeout service, and the memory management for parameter handling) and (iv) proposal of an efficient timeout service policy for event composition. This timeout policy reduces the number of messages sent from the timeout service to a
monitoring task to 1 per timeout irrespective of the number of event types in this monitoring task that are waiting for a timeout.

Contribution 5b: With respect to memory management for event composition, a novel combination of self-referencing memory and safe pointers, has shown that event composition scales better with respect to event type size ($\hat{e}$) than ordinary memory management. This combination also supports efficient distribution aspects such as aggregating several composite event occurrences into one message.

15.3 Future Work

The suggestions for future work presented here summarize ideas addressed throughout this Ph.D. thesis.

15.3.1 Comparing Event Specification Languages

It would be interesting to specify existing event specification languages using our schema for event composition to compare and contrast this with Solicitor or other event specification languages (an idea addressed in Section 6.5.7 on p. 148). In particular, it is desirable to see if such a comparison reveals more than the investigation by Motakis et al. [MZ97] and Zimmer et al. [ZU99]. For example, we have discovered similarities between event contexts that are not visible in any of these investigations. One critical issue is the need for a common policy for timestamping in the formalisms that are compared.

15.3.2 Automatic Translation to other Models

Since there are other formal models that offer model checking and simulation capability, it is interesting to automatically translate operator rules into such formal models [EA03, Eri02]. For example, by translating an event specification in, for example, Solicitor, into timed automata (based on the operator rules) it would be possible to use reachability analysis to analyze the event specification.

Also, it would be interesting to automatically translate the event operator rules into an implementation. For example, this would allow for specifying a new event context and automatically obtaining the proper algorithms for each operator in each event context that performs the actual event composition.
15.3 Future Work

15.3.3 Generic Event Specification Language

It would be interesting to specify a general event specification language meeting the following requirements (an idea brought up in Section 6.4 on p. 139): (i) it should encompass different timestamping policies, and (ii) it should have a complete set of event operators. Such a language would be interesting, since all other event specification languages would be special cases of this general event specification language. An advantage of this is, for example, that a property proven in the general event specification language can hold in the specific instances of event specification languages, unless the restriction of the specific language contradicts the property.

15.3.4 Adopt Time Constraint Preprocessing in Solicitor

Both Chronicle recognition [DGG93, Dou96] and monitoring of real-time logic [ML97a, ML97b] propose preprocessing of the event specifications (cf. Section 4.5.6 on p. 62) to remove unnecessary time constraints, detect impossible time constraints etc. It would be interesting to adopt these techniques into Solicitor, since this adoption may lead to more efficient event monitoring.

15.3.5 Hybrid Mechanism for Event Composition

Different mechanisms such as simple temporal networks used in Chronicle recognition [DGG93, Dou96] and graphs (similar to simple temporal networks) used in monitoring of real-time logic [ML97a, ML97b] have certain advantages over the mechanisms based on composing event occurrences out of contributing event occurrences, since both Chronicle recognition and monitoring of real-time logic supports a hybrid of forward and backward deduction (cf. Section 13.2.2 on p. 286). However, there are disadvantages, for example, both Chronicle recognition and monitoring of real-time logic are limited to certain event contexts. Therefore, it is interesting to investigate a hybrid event composer that uses the appropriate mechanism depending on the event specification.

15.3.6 Pruning of Event Occurrences

It is useful to enable grouping of event occurrences so that they can be removed with a single operation rather than one operation per event occurrence
Conclusions

(an idea brought up in Sections 10.6.2 to 10.6.3 on p. 235). For example, in network monitoring, many initiated composite event occurrences are never completed. Many event occurrences are removed without contributing to a composite event occurrence.

Grouping of event occurrences is used in Chronicle recognition. When a composite event occurrence is signalled, event monitoring removes all event occurrences that can only contribute to an overlapping event occurrence with respect to the signalled event occurrence. This is possible due to the use of a non-overlapping event context.

For overlapping event contexts, the issue is more complicated since signalling of one composite event occurrence does not automatically lead to removal of event occurrences. Rather, it is the timeout (derived from expiration times of currently initiated event compositions) that groups event occurrences together. Memory management that is segmented based on the timeouts would be interesting to investigate. The idea is that when a timeout occurs, all allocated memory in the segment is deallocated and the corresponding event occurrences are removed from event connector sets representing the filtered event log. The effect of grouping event occurrences based on timeouts depends on the application requirements as well as the time base of a system (i.e., the clock granularity employed in a system). With a sparse time base, the effect is conjectured to be significant.

15.3.7 Execution Time Analysis

A well-known problem in real-time systems is how to obtain worst-case execution times for tasks (cf. [Eng02]). There are solutions for hardware without pipelines or caches. However, for common processor architectures there are no complete models that scale well with respect to the size of an application. There are partial solutions to the problem.

Since event specification languages are declarative, it is conjectured that less effort is required to obtain the worst-case execution for event monitoring compared to an arbitrary program (an idea brought up in Section 10.11 on p. 237). The reason is that the algorithms performing the event composition do not change once they are correctly implemented; this fact can be used to transform a generic model of execution to a significantly simpler model applicable to event composition. For example, to give a complete model for event monitoring performed by dedicated processors would be an interesting topic of research, since the use of dedicated processors for event
monitoring is a viable solution and it simplifies the problem, because there is no interference of other tasks on the dedicated processor compared to the non-dedicated processors in the system.

One possibility is to create a simple model of event composition with assigned execution times (derived from the generic model) to each combination of event operator and event context. Given an event expression, it should be possible to derive the total execution time of event composition. Such a model could be based on refinements of the computational cost function in Ch. 9.

15.3.8 Architecture for Maintainable Efficient Event Composition

In this Ph.D. thesis, a monitor specification never changes during operation. However, there are several reasons why a monitor specification could change during operation. For example, in a highly available system such as phone switches we may want to monitor for composite events that were not declared in the monitor specification from the beginning. That is, during operation we want to add an event specification to the operational event monitoring service. The reason could be, for example, to debug a running application, add a new requirement, etc.

The major problem of on-line software updates is prevalent here, albeit more well-defined than for arbitrary code. This major problem is how the state of an (old) component is transferred to a new component (or the updated component). Given non-overlapping event contexts, it is possible to change an event specification after a composite event occurrence has been signalled since no overlapping results are allowed. However, for overlapping event contexts, the problem is more difficult. One idea here is to use the expiration time, an idea brought up in Section 10.3.1 on p. 230. When all potential composite event occurrences initiated before the update either have been completed and signalled, or have expired, then it is possible to switch from an old event specification to an updated event specification.

Another problem is the change in the resource requirements of a new component (or an updated component). Since event specification languages are declarative and more well-defined than arbitrary code, we conjecture that the problem is less complex. However, the question is if this difference in complexity is significant?
15.3.9 Recovery and Event Composition

One issue that has not been covered in this Ph.D. thesis is how database recovery affects event monitoring (a problem addressed in Section 10.3.2 on p. 231). For example, database recovery may imply that some transactions are re-executed, that partial effects of transactions are removed etc. First of all, the state of the event monitor is desirable to recover. Secondly, it is necessary to synchronize the state of the event monitor with the database; the expiration time of event types complicates the issue, since we are dealing with recovered event occurrences (that may have expired according to the current time) as well as new event occurrences.

One idea is that the database management system re-raises event occurrences while it reads the log. Another idea is that event monitoring is based on stable main memory and the database management system must resynchronize with the event monitor. In both cases, event occurrences can carry two timestamps (the original and the recovery timestamp) that are used for different situations. The potential solution to inform the event monitor of the difference between the current time and the original time during recovery will most likely not work, since the difference varies among event occurrences.

15.3.10 Automatic Generation of Event Monitor Specifications

Falkenroth et al. [FT99] address the problem of constructing predictable rule sets, and note that ECA rules are difficult, if not impossible, to analyze. Their approach is to take a higher-level specification that can be analyzed and generate the ECA-rules from this specification and, thereby, avoid the problems instead of analyzing a rule set to check if there are any problems. However, they do not address composite event types in their approach, since they are considered unnecessary. Similarly, Peters [Pet00] automatically generates real-time monitors from a system requirements specification.

Since event specifications themselves can be hard to analyze, it is desirable to consider an automatic generation of event specifications (an idea brought up in Section 14.2 on p. 300). For example, on-going work looks at how to translate timed automatas into ECA rules [EA03, Eri02]. Moreover, since Peters [Pet00] provides results concerning system requirements, it is valuable to demonstrate that it is possible to generate event monitor specifications from a system requirements specification.
15.3 Future Work

15.3.11 Providing Observability for Testing

An idea [Mel98b] (refined in [BMA99]) is to use event monitoring to provide observability while it is used for application purposes (briefly addressed in Section 3.3 on p. 32). This idea has been partially addressed (e.g., [LMA02, NAM02]). However, practical experiments are necessary to conduct in order to evaluate the usability of such an approach.

15.3.12 Relaxing the Requirements of Event Monitoring

Event monitoring has, in general, strict requirements on order. There are exceptions such as GEM [MSS97]. However, these are not completely convincing.

It would be interesting to consider relaxed requirements in an environment where the system requirements are formally specified and used to generate an event monitor (e.g., Peters [Pet00]) (an idea brought up in Section 7.3 on p. 174).

15.3.13 Parameterized Event Types

It is highly interesting to design an efficient implementation of connector sets for event types, parameterized on a parameter (an idea brought up in Section 14.1 on p. 297 (cf. Section 13.6.1 on p. 291)). For example, $E\langle type: tid, context: general \rangle = E_1;E_2$ (equivalent to the event expression $E = E_1;\langle ptype: tid, context: general \rangle E_2$) is a specification of an event type for each transaction identifier ($tid$) in a system; where the expression $E = (E_1;\langle context: general \rangle E_2)[E_1.tid = E_2.tid]\langle context: general \rangle$ is a definition of this specification, since it generates all combinations of $E_1$ and $E_2$ occurrences and then filters out those composite event occurrences whose contributing event occurrences do not have the same $tid$. It is not clear whether such a transformation is cost effective for practical sizes of filtered event logs, event type sizes, etc., in comparison to an event monitor where the connector sets are implemented as maps from a key (based on the parameter) to the event occurrences carrying the parameter. N.B., parameterized event specifications in other event contexts than the general cannot be expressed as a non-parameterized Solicitor event expression.

Given parameterized event types, then the monitor specification shown in Fig. 10.4 on p. 229 can be reduced to Fig. 15.1 on the following page. N.B., the negating contributing event types in a non-occurrence expression (e.g.,
monitor ElevatorMonitor {
  primitive {
    event floor(ptype: level)(int level, int card).
    event arrive(ptype: level)(int level).
    event emergency.
  }
  task taskOne(context: chronicle) {
    event correct(ptype: level)(int level, int card) is
      N(expire: 10min)(floor, emergency, arrive).
    event incorrect(ptype: level)(int level, int card) is
      N(floor, arrive, floor+10min).
  }
}

Figure 15.1: Parameterized event monitor specification example

the arrive event type in the incorrect event expression) are not restricted
by the parameter equivalence specified by ptype in this case. To handle
parameterized event types, both this case and the case where the negating
event type should follow the parameterization restriction must be handled.

Other attributes such as expiration time and order of event occurrences
need to be addressed in parameterized event types. The order of occurrence
can perhaps be managed by providing parameterization with respect to the
contributed event type. For example,

\[ E = E_1; (context: general, ptype: left.order = right.order + 1)E_1 \]

means that the right operand is a parameterized event type with respect to
each event occurrence of the right operand. A different syntax would be to
use a function, similar to the monitoring of real-time logic. For example,
assume that \( \bullet(E, i) \) means the occurrence of \( E \) with relative index \( i \), then
we could specify

\[ E = \bullet(E_1, i); \bullet(E_1, i+1) \]

Yet another possibility would be to state it as an attribute:

\[ E = E_1(order: i); E_1(order: i+1) \]
Part V

Bibliography and Appendices
References


References


References

Joint Conference on Artificial Intelligence [IJCAI 93], pages 166–172, 1993.


References


References


[PS02] P. Pietzuch and B. Shand. A framework for object-based event composition in distributed systems. In 12th Workshop for PhD Students in Object-Oriented Systems (PhDOOS’02), Malaga, Spain, June 2002.

[Ron97] C. L. Roncancio. Toward duration-base, constrained and dy-
namic event types. In Proc. 2nd Int’l Workshop on Active, Real-
To be published.

[Sch94a] F. B. Schneider. Replication management using the state-
machine approach. In Sape Mullender, editor, Distributed sys-

[Sch94b] W. Schütz. Fundamental issues in testing distributed real-time

[Sch95a] B. A. Schroeder. On-line monitoring: A tutorial. Computer,


[Sch96] S. Schwidersky. Monitoring the Behaviour of Distributed Sys-
tems. PhD thesis, Selwyn College, University of Cambridge,
1996.


[SHH87] M. Stonebraker, E. Hanson, and C. H. Hong. The design of the
POSTGRES rules system. In Proc. 3rd IEEE, pages 365–374,
December 1987.


[Sno88] R. Snodgrass. A relational approach to monitoring complex sys-

[Sta95] J. Stankovic. Adjustable flow control filters, reflective memories,
and coprocessors as support for distributed real-time systems.

[TM89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A distrib-
uted real-time kernel. ACM SIGOPS Operating Systems Revies,


Appendix A

Formal Notation Convention and Symbol Dictionary

This appendix contains the formal notation convention used throughout this Ph.D. thesis as well as a symbol dictionary.

A.1 Formal Notation Convention

The formalism in this Ph.D. thesis is based on set theory. Two types of sets are used: static and dynamic sets. A static set never changes (at least not during event composition) and are used to denote sets of everything (e.g., the set of all natural numbers). The blackboard alphabet is used to denote such sets (e.g., $\mathbb{N}, \mathbb{R}$). A dynamic set can change during event composition and are denoted by calligraphic symbols (e.g., $\mathcal{C}, \mathcal{I}$). To distinguish a set before and after a change, we add a citation mark on the symbol to denote each change. For example, if we want to add an element to $\mathcal{C}$, then this can be written $\mathcal{C}' = \mathcal{C} \cup \{\text{element}\}$, where $\mathcal{C}'$ is the result of the change and $\mathcal{C}$ is the set before the change.

First-order predicate logic is used to define the formalism. Lower-case italicized symbols are either functions, predicates, atoms, or variables. The variables $i, j, k, l, m, n$ are always in $\mathbb{N}$. Upper-case italicized symbols refer to expressions and constants. Greek symbols are used for representations of event occurrences and functions when the full function name is too long. The logic relations $\land$, $\lor$, $\otimes$, $\Rightarrow$, $\Leftrightarrow$, and $\neg$ stand for “and”, “or”, “exclusive or”, “implies”, “equivalent to” and “negation”.

335
The quantifies are written \('∃' and '∀' meaning “there exists” and “for all”\. Axioms and definitions are written as: quantified variables \(P\) meaning “for the quantified variables \(P\) holds”. For example, \(∀x ∈ \mathbb{N}(x > 0)\) is read “for all \(x\) bound to a natural number \(\mathbb{N}\) \(x > 0\) holds”.

\(X \overset{\Delta}{=} Y\) stands for \(X\) is a lexical substitution for \(Y\). Basically, if the left side is found in a predicate, then it can be replaced with the right side. A lexical substitution has no unbound variables with respect to the predicate it is used in. For example, we cannot state \(a(x) \overset{\Delta}{=} b(x) \Rightarrow c(x)\) in \(P\).

We use \(\overset{\Delta}{=}\) to define predicates in terms of other predicates, where

\[
∀\text{unbound variables that appears both in } P \text{ and } Q ((P \Leftrightarrow Q)) \Leftrightarrow P \overset{\Delta}{=} Q
\]

holds. For example, \(married(x) \overset{\Delta}{=} male(x) \land adult(x) \land ¬bachelor(x)\) is the same thing as \(∀x(married(x) \Leftrightarrow male(x) \land adult(x) \land ¬bachelor(x))\).

In proof outlines (of code), the \{ and \} enclose predicates that should hold between the statements. In other literature, \{ and \} is used. However, to avoid confusion with the set notation, \}} and {{{{{ has been chosen instead.

## A.2 Symbol Dictionary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{A} \alpha )</td>
<td>With respect to a context predicate, all non-expired event occurrences that are of any event type in ( \mathcal{E}_\alpha ), in Def. 6.36 on p. 128.</td>
</tr>
<tr>
<td>( \mathcal{A} \omega )</td>
<td>With respect to a context predicate, all non-expired event occurrences that are of any event type in ( \mathcal{E}_\omega ), in Def. 6.36 on p. 128.</td>
</tr>
<tr>
<td>all_of_types(( \mathcal{E} ))</td>
<td>All non-expired event occurrences of a set of event types ( (\mathcal{E}) ), in Def. 6.35 on p. 128.</td>
</tr>
</tbody>
</table>

\(\text{continued on the following page}\)
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_operands((E))</td>
<td>Returns all immediate contributing event types of the principal event operator of the event expression defining event type (E), in Def. 6.17 on p. 114.</td>
</tr>
<tr>
<td>all_operands((E))</td>
<td>Returns all primitive contributing event types of the event expression defining event type (E), in Def. 6.17 on p. 114.</td>
</tr>
<tr>
<td>(\alpha(E))</td>
<td>Returns all immediate contributing event types of the principal event operator of the event expression defining (E) that can initiate an event occurrence of (E), in Def. 6.17 on p. 114 and Def. 6.30 on p. 125.</td>
</tr>
<tr>
<td>(\hat{\alpha}(E))</td>
<td>Returns all primitive contributing event types of the event expression defining (E) that can initiate an event occurrence of (E), in Def. 6.17 on p. 114 and Def. 6.31 on p. 126.</td>
</tr>
<tr>
<td>(\alpha^{\uparrow}(\gamma))</td>
<td>The initiating event occurrence of an immediate contributing event type of the principal event operator of the event expression defining the event type of (\gamma), in Def. 6.22 on p. 122.</td>
</tr>
<tr>
<td>(\hat{\alpha}^{\uparrow}(\gamma))</td>
<td>The ultimate initiating event occurrence of an contributing primitive event type of the principal event operator of the event expression defining the event type of (\gamma), in Def. 6.23 on p. 122.</td>
</tr>
<tr>
<td>(chronicle(E, \gamma, \gamma, E, E))</td>
<td>The chronicle context predicate, in Def. 6.39 on p. 129.</td>
</tr>
<tr>
<td>(condition(s, r, E))</td>
<td>A condition that is true if rule (r) applied to (E) is true in state (s), in Def. 6.11 on p. 109 and Defs 6.15 to 6.16 on p. 113.</td>
</tr>
</tbody>
</table>

continued on the following page
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>conclusion(s, r, E)</td>
<td>A function returning a new state $s'$ derived from $s$ by applying rule $r$ on event type $E$, in Def. 6.11 on p. 109 andDefs 6.15 to 6.16 on p. 113.</td>
</tr>
<tr>
<td>$\mathcal{CONS}_\alpha$</td>
<td>With respect to a context predicate, all event occurrences that has been consumed as initiators with respect to $E$, in Def. 6.38 on p. 128.</td>
</tr>
<tr>
<td>$\mathcal{CONS}_\omega$</td>
<td>With respect to a context predicate, all event occurrences that has been consumed as terminators with respect to $E$, in Def. 6.38 on p. 128.</td>
</tr>
<tr>
<td>$continuous(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega)$</td>
<td>The continuous context predicate, in Def. 6.41 on p. 130.</td>
</tr>
<tr>
<td>com($X$)</td>
<td>Returns the set of composite event occurrence of $X = {\gamma', \gamma}$, in Def. 6.2 on p. 104.</td>
</tr>
<tr>
<td>$\mathcal{COMP}$</td>
<td>With respect to a context predicate, all event occurrences that has been composed of an event type $E$, in Def. 6.37 on p. 128.</td>
</tr>
<tr>
<td>$c_{ord}(E)$</td>
<td>The conflict order of $E$ used to arbitrate among simultaneous event occurrences of related event types such as event types that contribute to the same composite event type, in Def. 6.19 on p. 120.</td>
</tr>
<tr>
<td>end([t, t'])</td>
<td>A function that returns the end of the interval $[t, t'] (t')$, in Def. 4.14 on p. 53.</td>
</tr>
<tr>
<td>expired($\gamma$)</td>
<td>A predicate that is true if the now minus the start of $\gamma$ is larger than the expiration time of the event type of $\gamma$, in Def. 6.27 on p. 124.</td>
</tr>
<tr>
<td>$E$</td>
<td>An event type, in Section 4.2 on p. 42.</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>A set of event types.</td>
</tr>
</tbody>
</table>

*continued on the following page*
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_\alpha$</td>
<td>A set of event types whose occurrences can contribute as initiators to a composite, in Section 6.3.4 on p. 126.</td>
</tr>
<tr>
<td>$E_\omega$</td>
<td>A set of event types whose occurrences can contribute as terminators to a composite, in Section 6.3.4 on p. 126.</td>
</tr>
<tr>
<td>$E_m$</td>
<td>The set of monitored event types, in Section 9.1 on p. 197.</td>
</tr>
<tr>
<td>$\hat{e}$</td>
<td>The size of an event expression. Used in the time complexity investigation, in Section 9.2 on p. 201.</td>
</tr>
<tr>
<td>$G$</td>
<td>The set of generated event occurrences up until now, in Axiom 4.1 on p. 53.</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>A variable representing an event occurrence, in Section 4.5.1 on p. 52.</td>
</tr>
<tr>
<td>$\langle \gamma', \gamma \rangle$</td>
<td>A binary tuple where $\gamma'$ either has initiated or terminated $\gamma$, in Section 6.2 on p. 102.</td>
</tr>
<tr>
<td>${\langle \gamma', \gamma \rangle}$</td>
<td>A set of $\langle \gamma', \gamma \rangle$, in Section 6.2 on p. 102.</td>
</tr>
<tr>
<td>$\Gamma(E, [t, t'])$</td>
<td>A relation describing that event type $E$ has occurred throughout the interval $[t, t']$, in Section 4.5.1 on p. 52.</td>
</tr>
<tr>
<td>$\hat{g}$</td>
<td>Size of filtered event log (event history) used in time complexity investigation, in Section 9.2 on p. 201.</td>
</tr>
<tr>
<td>$\text{general}(E, \gamma_\alpha, \gamma_\omega, E_\alpha, E_\omega)$</td>
<td>The general (unconstrained) context predicate, in Def. 6.42 on p. 130.</td>
</tr>
<tr>
<td>$i$</td>
<td>Out of $g$, the number of initiating event occurrences, in Section 9.2 on p. 201.</td>
</tr>
<tr>
<td>$\text{iot}(\mathcal{X})$</td>
<td>Returns the set of initiators or terminators (depending on what $\mathcal{X} = {\langle \gamma', \gamma \rangle}$ represents, in Def. 6.1 on p. 104.</td>
</tr>
<tr>
<td>$I$</td>
<td>Used to represent a set of invalidated event occurrences.</td>
</tr>
</tbody>
</table>

continued on the following page
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_α(E)$</td>
<td>The set of invalidated occurrences that cannot contribute to a composite event occurrence of type $E$ as an initiator in the future, in Section 6.2 on p. 102.</td>
</tr>
<tr>
<td>$I_ω(E)$</td>
<td>The set of invalidated occurrences that cannot contribute to a composite event occurrence of type $E$ as a terminator in the future, in Section 6.2 on p. 102.</td>
</tr>
<tr>
<td>$INVαC_α$</td>
<td>With respect to a context predicate, equivalent to $I_α(E)$, in Def. 6.37 on p. 128.</td>
</tr>
<tr>
<td>$INVωC_ω$</td>
<td>With respect to a context predicate, equivalent to $I_ω(E)$, in Def. 6.37 on p. 128.</td>
</tr>
<tr>
<td>$later_state(s, s')$</td>
<td>True if $s$ is a later state than $s'$, in Def. 6.9 on p. 108.</td>
</tr>
<tr>
<td>$now$</td>
<td>The current time, in Section 4.2 on p. 42.</td>
</tr>
<tr>
<td>$ω(E)$</td>
<td>Returns all immediate contributing event types of the principal event operator of the event expression defining $E$ that can terminate an event occurrence of $E$, in Def. 6.17 on p. 114 and Def. 6.32 on p. 126.</td>
</tr>
<tr>
<td>$\hat{ω}(E)$</td>
<td>Returns all primitive contributing event types of the event expression defining $E$ that can terminate an event occurrence of $E$, in Def. 6.17 on p. 114 and Def. 6.33 on p. 126.</td>
</tr>
<tr>
<td>$ω↑(γ)$</td>
<td>The terminating event occurrence of an immediate contributing event type of the principal event operator of the event expression defining the event type of $γ$, in Def. 6.20 on p. 121.</td>
</tr>
</tbody>
</table>

continued on the following page
## A.2 Symbol Dictionary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\gamma}^+$</td>
<td>The ultimate terminating event occurrence of an contributing primitive event type of the principal event operator of the event expression defining the event type of $\gamma$, in Def. 6.21 on p. 122.</td>
</tr>
<tr>
<td>$\bigcirc{\mathcal{P}}_R$</td>
<td>The temporal ordering operators applicable to event occurrences, in Section 4.5.1 on p. 52.</td>
</tr>
<tr>
<td>$\hat{p}$</td>
<td>Size of parameters used in the time complexity investigation, in Section 9.2 on p. 201.</td>
</tr>
<tr>
<td>$P_i$</td>
<td>The processing tuple for an event type defining state $s_i$, in Section 6.2 on p. 102.</td>
</tr>
<tr>
<td>$\mathcal{P}_i$</td>
<td>A set of processing tuples (one for each event type) defining state $s_i$, in Section 6.2 on p. 102.</td>
</tr>
<tr>
<td>$\text{prim}(E)$</td>
<td>Predicate that is true if $E$ is a primitive, in Section 4.2 on p. 42.</td>
</tr>
<tr>
<td>$Q_E(E')$</td>
<td>A connector set between $E'$ (contributing event type) and $E$ (the contributed event type), in Section 7.1 on p. 152.</td>
</tr>
<tr>
<td>recent($E, \gamma_\alpha, \gamma_\omega, \xi_\alpha, \xi_\omega$)</td>
<td>The recent context predicate, in Def. 6.40 on p. 129.</td>
</tr>
<tr>
<td>rules($E$)</td>
<td>Returns rules applicable to event type $E$, in Def. 6.10 on p. 108.</td>
</tr>
<tr>
<td>$s$</td>
<td>Represent a state defined as the tuple $\langle G_i, \mathcal{P}_i \rangle$, in Section 6.2 on p. 102.</td>
</tr>
<tr>
<td>span($\gamma$)</td>
<td>A function that returns the span (interval) of the event occurrence in the variable $\gamma$, in Def. 4.13 on p. 52.</td>
</tr>
<tr>
<td>start([$t, t'$])</td>
<td>A function that returns the start time of the interval $[t, t']$ ($t$), in Def. 4.14 on p. 53.</td>
</tr>
<tr>
<td>$t$</td>
<td>Denotes a time, in Section 4.2 on p. 42.</td>
</tr>
<tr>
<td>$T$</td>
<td>Denotes a duration, in Section 4.2 on p. 42.</td>
</tr>
</tbody>
</table>

continued on the following page
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>([t, t'])</td>
<td>Denotes an interval between (t) and (t'), in Section 4.2 on p. 42.</td>
</tr>
<tr>
<td>(t)</td>
<td>Same as (\hat{t}) for terminators, in Section 9.2 on p. 201.</td>
</tr>
<tr>
<td>(\text{type}(\gamma))</td>
<td>A function that returns the event type of the occurrence in the variable (\gamma), in Def. 4.13 on p. 52.</td>
</tr>
<tr>
<td>(U)</td>
<td>A variable representing event occurrences used as initiators or terminators. It is a set of (\langle \gamma, \gamma' \rangle).</td>
</tr>
<tr>
<td>(U_\alpha(E))</td>
<td>The set of initiator occurrences used to contribute to a composite event occurrences. It is a set of (\langle \gamma, \gamma' \rangle). Each operator has its own set. Defined in Section 6.2 on p. 102</td>
</tr>
<tr>
<td>(U_\omega(E))</td>
<td>As (U_\alpha), but for terminator occurrences instead of initiator occurrences. Defined in Section 6.2 on p. 102</td>
</tr>
<tr>
<td>(\text{UNCONS}_\alpha)</td>
<td>With respect to a context predicate, all event occurrences that are unconsumed initiators with respect to (E), in Def. 6.38 on p. 128.</td>
</tr>
<tr>
<td>(\text{UNCONS}_\omega)</td>
<td>With respect to a context predicate, all event occurrences that are unconsumed terminators with respect to (E), in Def. 6.38 on p. 128.</td>
</tr>
<tr>
<td>(\text{USED}_\alpha)</td>
<td>With respect to a context predicate, equivalent to (U_\alpha(E)), in Def. 6.37 on p. 128.</td>
</tr>
<tr>
<td>(\text{USED}_\omega)</td>
<td>With respect to a context predicate, equivalent to (U_\omega(E)), in Def. 6.37 on p. 128.</td>
</tr>
<tr>
<td>(\text{VALID}_\alpha)</td>
<td>With respect to a context predicate, all event occurrences that are valid initiators with respect to (E), in Def. 6.38 on p. 128.</td>
</tr>
</tbody>
</table>

*continued on the following page*
### A.2 Symbol Dictionary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{V}<em>{\text{ALTD}</em>\omega}$</td>
<td>With respect to a context predicate, all event occurrences that are valid terminators with respect to $E$, in Def. 6.38 on p. 128.</td>
</tr>
<tr>
<td>$X(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}<em>\alpha, \mathcal{E}</em>\omega)$</td>
<td>Generic event context predicate that is true for an initiator occurrence $\gamma_\alpha$ and a terminator occurrence $\gamma_\omega$ for the event type $E$ where $\mathcal{E}<em>\alpha \subseteq \alpha(E)$ and $\mathcal{E}</em>\omega \subseteq \omega(E)$, in Def. 6.34 on p. 127.</td>
</tr>
<tr>
<td>$X_\alpha(E, \gamma_\alpha, \mathcal{E}_\alpha)$</td>
<td>Part of the generic event context predicate that is true for an initiator occurrence $\gamma_\alpha$ for the event type $E$ where $\mathcal{E}_\alpha \subseteq \alpha(E)$, in Def. 6.34 on p. 127.</td>
</tr>
<tr>
<td>$X_\omega(E, \gamma_\omega, \mathcal{E}_\omega)$</td>
<td>Part of the generic event context predicate that is true for a terminator occurrence $\gamma_\omega$ for the event type $E$ where $\mathcal{E}_\omega \subseteq \omega(E)$, in Def. 6.34 on p. 127.</td>
</tr>
<tr>
<td>$X_{\alpha\omega}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}<em>\alpha, \mathcal{E}</em>\omega)$</td>
<td>Part of the generic event context predicate that is true for an appropriate relation between an initiator occurrence $\gamma_\alpha$ and a terminator occurrence $\gamma_\omega$ for the event type $E$ where $\mathcal{E}<em>\alpha \subseteq \alpha(E)$ and $\mathcal{E}</em>\omega \subseteq \omega(E)$, in Def. 6.34 on p. 127.</td>
</tr>
</tbody>
</table>
Formal Notation Convention and Symbol Dictionary
This appendix contains the complete set of operator rules for Solicitor. The proof techniques use the notation for permutations presented in Section 6.3.5 on p. 135.

B.1 Solicitor Syntax

B.1.1 The Complete Event Expression Syntax

Definition B.1
Complete event expression syntax:
Let \( E_P \in \{ E | \text{prim}(E) \} \) in

\[
E ::= E_P \mid E \langle \text{Attributes} \rangle \mid E (\uparrow \nabla \Delta \llbracket \text{Attributes} \rrbracket) E \mid (A | CA | N) \langle \text{Attributes} \rangle (E, E, E') \mid E + \langle \text{Attributes} \rangle T \mid E (![\llbracket Q \rrbracket]) \langle \text{Attributes} \rangle (E, E')
\]

\[
Q ::= QC \parallel QC \langle Q' \rangle
\]

\[
QC ::= QD \&\& QC \langle QD \rangle \langle Q' \rangle
\]

\[
QD ::= \text{Attribute} \equiv (\text{value} | \text{Attribute}) | \text{true} | \text{false}
\]
\[
\text{Attributes ::= ' < EncAttributes '>' }
\]
\[
\text{EncAttributes ::= Attribute : value[, EncAttributes]}
\]
\[
\text{Attribute ::= (value|Attribute[, EncAttributes]|ε}
\]
\[
\text{Attribute ::= identity|identity(identity)}
\]

**B.1.2 The Event Monitor Specification Syntax**

The following four definitions are copies of Defs 10.1 to 10.4 on p. 228:

**Definition 10.1**

Event monitor specification:

\[
\text{Monitor ::= monitor identity [Attributes] '{ EventGroupings '} }
\]

\[
\text{EventGroupings ::= (EMTask|PrimitiveEvents) +}
\]

**Definition 10.2**

Event monitoring task:

\[
\text{EMTask ::= task identity [Attributes] '{ EventSpecification * '} }
\]

\[
\text{EventSpecification ::= event identity[Attributes][(' ParSpec ')] is E '}
\]

**Definition 10.3**

Primitive event specification:

\[
\text{PrimitiveEvents ::= primitive { PrimitiveEvent*}'}
\]

\[
\text{PrimitiveEvent ::= event identity[(' ParSpecs ')].'}
\]
B.2 Basic Definitions

Definition 10.4
Parameter specifications:
\[ ParSpecs ::= ParSpec \mid ParSpecs \epsilon \]
\[ ParSpec ::= (\text{int} | \text{double} | \text{char}) \text{ identity } [\text{integerConstant}] \]

B.2 Basic Definitions

Definition 4.13
Occurrence functions:
Let \( \gamma \vdash \Gamma(E, [t,t']) \) in
\[ \text{type}(\gamma) = E \]
\[ \text{span}(\gamma) = [t, t'] \]

\[ \square \]

Definition 4.14
Interval functions:
Let \([t, t']\) be an interval in
\[ \text{start}([t, t']) = t \]
\[ \text{end}([t, t']) = t' \]
\[ ||[t, t']|| = t' - t \]

\[ \square \]

Axiom 4.14
Occurrence history:
\[ G \subseteq T \land \forall \gamma \in G \ (\text{end}(\text{span}(\gamma)) \leq \text{now}) \]

\[ \square \]
Definition 6.1
Initiator or terminator-set-of function:
Let $\mathcal{U} = \{\langle \gamma', \gamma \rangle\}$ (where $\mathcal{U}$ is any set of initiator/terminator tuples), then

$$\text{iot}(\mathcal{U}) = \{\gamma' | \langle \gamma', \gamma \rangle \in \mathcal{U}\}$$

□

Definition 6.2
Composite-set-of function:
Let $\mathcal{U} = \{\langle \gamma', \gamma \rangle\}$, then

$$\text{com}(\mathcal{U}) = \{\gamma | \langle \gamma', \gamma \rangle \in \mathcal{U}\}$$

□

Definition 6.3
Initiator predicate:

\[\text{initiate}(\gamma_1, \gamma_2) \trianglerighteq \langle \gamma_1, \gamma_2 \rangle \in \mathcal{U}_\alpha(\text{type}(\gamma_2))\]

□

Definition 6.4
Terminator predicate:

\[\text{terminate}(\gamma_1, \gamma_2) \triangleq (\gamma_1, \gamma_2) \in \mathcal{U}_\omega(\text{type}(\gamma_2))\]

□
Definition 6.5
Event occurrence set relations:

\[
\begin{align*}
X \sqsubset X' & \triangleq X \subseteq X' \land \\
& \forall \gamma, \gamma' \in X \land \gamma' \in (X' \setminus X) \Rightarrow \text{end(span}(\gamma)) \leq \text{end(span}(\gamma'))) \\
X \sqsubseteq X' & \triangleq X \subseteq X' \lor X = X' \\
X \sqsupset X' & \triangleq X' \sqsubseteq X \\
X \sqsupseteq X' & \triangleq X' \subseteq X
\end{align*}
\]

\[\square\]

Definition 6.6
Processing tuple is of event type predicate:

\[
event\_type\_of\ (P, E) \triangleq \forall E', U, U', I, I' (P = (E', U, U', I, I') \Rightarrow E = E')
\]

\[\square\]

Definition 6.7
Successive processing tuples for the same event type:

\[
\begin{align*}
later\_processing\_tuple\_state\ (P_i, P_j) & \triangleq \\
& \forall E, U_i, U'_i, I_i, I'_i, U_j, U'_j, I_j, I'_j \\
& \langle (E, U_i, U'_i, I_i, I'_i) = P_i \land (E, U_j, U'_j, I_j, I'_j) = P_j \Rightarrow \\
& \left( U_i \sqsupset U_j \land U'_i \sqsupset U'_j \land I_i \sqsupset I_j \land I'_i \sqsupset I'_j \right) \lor \\
& \left( U_i \sqsupset U_j \land U'_i \sqsupset U'_j \land I_i \sqsupset I_j \land I'_i \sqsupset I'_j \right) \lor \\
& \left( U_i \sqsupset U_j \land U'_i \sqsupset U'_j \land I_i \sqsupset I_j \land I'_i \sqsupset I'_j \right) \lor \\
& \left( U_i \sqsupset U_j \land U'_i \sqsupset U'_j \land I_i \sqsupset I_j \land I'_i \sqsupset I'_j \right)
\end{align*}
\]

\[\square\]
Definition 6.8
Same processing tuple for an event type:

\[ \text{same\_processing\_tuple\_state}(P_i, P_j) \overset{\Delta}{=} \forall E, U_i, U'_i, T_i, T'_i, U_j, U'_j, T_j, T'_j \]
\[ \left( \langle E, U_i, U'_i, T_i, T'_i \rangle = P_i \land \langle E, U_j, U'_j, T_j, T'_j \rangle = P_j \implies \right. \]
\[ \left( U_i = U_j \land U'_i = U'_j \land T_i = T_j \land T'_i = T'_j \right) \]

\[ \square \]

Definition 6.9
Later state than:

\[ \text{later\_state}(s, s') \overset{\Delta}{=} \forall G_i, G_j, P_i, P_j \]
\[ (s = \langle G_i, P_i \rangle \land s' = \langle G_j, P_j \rangle \implies \]
\[ G_i \sqsubseteq G_j \lor \]
\[ G_i = G_j \land \]
\[ \exists E \forall E' (E \neq E' \implies \]
\[ (\forall P_i, P'_i, P_j, P'_j \]
\[ (P_i \in P_i \land \text{event\_type\_of}(P_i, E) \land \]
\[ \text{event\_type\_of}(P'_i, E') \land \]
\[ P_j \in P_j \land \text{event\_type\_of}(P_j, E) \land \]
\[ \text{event\_type\_of}(P'_j, E') \implies \]
\[ \text{same\_processing\_tuple\_state}(P_i, P_j) \land \]
\[ (\text{same\_processing\_tuple\_state}(P'_i, P'_j) \lor \]
\[ \text{same\_processing\_tuple\_state}(P'_i, P'_j))))))]) \]

\[ \square \]

Definition 6.10
Rules function:

\[ \text{rules}(E) \text{ is a function that returns the set of rules } \{\{r\}\} \text{ applicable to } E. \]

\[ \square \]
Definition 6.11

Least constrained event composition: This algorithm is a refinement of the algorithm presented by Nilsson [Nil82, p. 21] depicted in Fig. 6.2 on p. 101. Let $s$ be the state of event composition; let the condition$(s, r, E)$ predicate be true if the condition of rule $r$ evaluated on event type $E$ is true in $s$ (precise definition in Defs 6.15 to 6.16 on p. 113); let conclusion$(s, r, E)$ return a state change (precise definition in Defs 6.15 to 6.16 on p. 113), let $I \triangleq (\text{later}_\text{state}(s, s_0) \lor s=s_0)$, and let $C \triangleq r \in \text{rules}(E) \land \text{condition}(s, r, E)$ then

\[
\begin{align*}
\{I\} \\
\text{while } \exists E, r (C) \text{ do begin} \\
\quad \{I \land \exists E, r (C)\} \\
\quad \text{select } E, r \text{ such that } C \\
\quad \{I \land C\} \quad /\ast C \Rightarrow \exists E, r (C) \ast/ \\
\quad s := \text{conclusion}(s, r, E); /\ast \text{ updates state } s \ast/ \\
\quad \{I \land \text{later}_\text{state}(s, s')\} \quad /\ast \exists E, r (C) \text{ is undefined, } s' \text{ is earlier state } \ast/ \\
\text{end} \\
\{I \land \neg \exists E, r (C)\}
\end{align*}
\]

\[\square\]

Definition 6.12

Usage rule form:

Let $R$ be the range (or domain) predicate that ensures that the event occurrences in the candidate event occurrences are of specific event types, and $R_c$ be the context predicate that ensures that candidate event occurrences can be used (or invalidated) according to the event context; let $H$ be the hypothesis of the rule addressing the operator semantics (e.g., in Defs 4.3

\[\text{The } C \text{ (a lexical substitution) is used in two different ways in this definition. Firstly, it is used in the while statement where } E \text{ and } r \text{ are properly bound. Secondly, it is used in the predicates between statements where } E \text{ and } r \text{ are bound to the variables declared in the algorithm.}\]

\[1\text{The } C \text{ (a lexical substitution) is used in two different ways in this definition. Firstly, it is used in the while statement where } E \text{ and } r \text{ are properly bound. Secondly, it is used in the predicates between statements where } E \text{ and } r \text{ are bound to the variables declared in the algorithm.}\]
to 4.11 on pp. 47–49); let $C$ be the conclusion written as a sequence of assignments of sets ($\mathcal{X}$) updated as $\mathcal{X} \cup= \Delta \mathcal{X}$ (e.g., $\mathcal{G} \cup= \Delta \mathcal{G}$); then the usage rule form is written as:

$$\forall E \forall \gamma_1, \ldots, \gamma_n \in \mathcal{G} :$$

\[
\begin{array}{c}
R \\
R_c \\
H \\
\hline
C
\end{array}
\]

This is equivalent to $\forall E \forall \gamma_1, \ldots, \gamma_n \in \mathcal{G} (R \land R_c \land H \rightarrow C)$.

□

**Definition 6.13**

**Invalidation rule form:**

Using the same assumptions for the usage rule form and let $H'$ be the invalidation hypothesis such that $\forall E \forall \gamma_1, \ldots, \gamma_m \in \mathcal{G} (R \Rightarrow (\neg H' \Rightarrow H))$; let $BVIC = \{\gamma_i, \ldots, \gamma_m\} \cap \{\gamma_j, \ldots, \gamma_p\} = \emptyset$ be the bound variable interrelation constraint, and $R'_c$ is either the part of the event context that is associated with the initiator or the terminator, then the invalidation rule form is written as:

$$\forall E \forall \gamma_i, \ldots, \gamma_n \in \mathcal{G} :$$

\[
\begin{array}{c}
R_{in} \\
R'_c \\
\forall \gamma_j, \ldots, \gamma_p \in \mathcal{G} (R_{jp} \land BVIC \land R_c \Rightarrow H') \\
\hline
C
\end{array}
\]

An equivalent form is $\forall E \forall \gamma_i, \ldots, \gamma_n \in \mathcal{G} (R_{in} \land R'_c \forall \gamma_j, \ldots, \gamma_p \in \mathcal{G} (R_{jp} \land BVIC \land R_c \Rightarrow H') \rightarrow C)$.

□
Definition 6.14
Apply conclusion function:
The function \( s = \text{apply}(C, s', E) \) reads “apply assignments in \( C \) to state \( s' \) for event type \( E \) and return the new state \( s \).” □

Definition 6.15
Usage rule form predicates:
Let \( s \) be a state, and let \( r \triangleq \forall E \forall \gamma_1, \ldots, \gamma_n \in G (R \land R_c \land H \rightarrow C) \), then
\[
\text{condition}(s, r, E) \triangleq r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G (R \land R_c \land H) \\
\text{range}(s, r, E) \triangleq r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G (R \land R_c) \\
\text{conclusion}(s, r, E) \triangleq \text{apply}(C, s, E)
\]
□

Definition 6.16
Invalidation rule form predicates:
Let \( s \) be a state, and let \( r \triangleq \forall E \forall \gamma_1, \ldots, \gamma_n \in G (R_{in} \land R_{c}^j \land \forall \gamma_j, \ldots, \gamma_p \in G (R_{jp} \land BV IC \land R_c \Rightarrow H') \rightarrow C) \), then
\[
\text{condition}(s, r, E) \triangleq r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G (R_{in} \land R_{c}^j \land \forall \gamma_j, \ldots, \gamma_p \in G (R_{jp} \land BV IC \land R_c \Rightarrow H')) \\
\text{range}(s, r, E) \triangleq r \in \text{rules}(E') \land \exists \gamma_1, \ldots, \gamma_n \in G (R_{in} \land R_{c}^j \land \forall \gamma_j, \ldots, \gamma_p \in G (R_{jp} \land BV IC \land R_c \Rightarrow true)) \\
\text{conclusion}(s, r, E) \triangleq \text{apply}(C, s, E)
\]
□
Definition 6.17
Event Expression Property Functions for an event specification language:
To be more specific, it is necessary to define functions for an event specification language that:

1. for the principal event operator of an event type
   
   (a) all operands$(E)$: return a set of all immediate contributing event types (e.g., $\{E_1, E_2\}$ are the immediate contributing event types of $E_1; E_2$ regardless of whether $E_1$ and $E_2$ are primitive or composite
   
   (b) $\alpha(E)$: return all immediate potential initiating contributing event types (e.g., the $E_1$ is an immediate potential initiating contributing event type of $E_1; E_2$)
   
   (c) $\omega(E)$: return all immediate potential terminating contributing event types (e.g., the $E_2$ is an immediate potential terminating contributing event type of $E_1; E_2$

2. for the whole event expression defining an event type

   (a) all operands$(E)$: return a set of all primitive contributing event types (e.g., $\{E_1, E_2, E_3\}$ is the set of all primitive contributing event types of $(E_1; E_2); E_3$)
   
   (b) $\hat{\alpha}(E)$: return a set of all primitive potential initiating contributing event types (e.g., $\{E_1\}$ is the set of all primitive potential initiating contributing event types of $(E_1; E_2); E_3$)
   
   (c) $\hat{\omega}(E)$: return a set of all primitive potential terminating contributing event types (e.g., $\{E_3\}$ is the set of all primitive potential terminating contributing event types of $(E_1; E_2); E_3$)
Definition 6.19
Conflict order arbitration of event types:
\( c_{\text{ord}}(E) \) returns the order of event type \( E \), where \( c_{\text{ord}}(E) \leq c_{\text{ord}}(E') \) iff event type \( E \) is of lower or equal order compared to event type \( E' \).

\[ \square \]

Definition 6.20
Immediate terminator occurrence: Let \( \gamma \triangleq \Gamma(E, [t, t']) \in \mathcal{G} \), then
\[
\omega\uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff } \text{prim}(E) \\
\gamma' \text{ where } \gamma' \in \{ \gamma'' | \text{terminate}(\gamma'', \gamma) \} & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

\[ \square \]

Definition 6.21
Ultimate terminator occurrence: Let \( \gamma \triangleq \Gamma(E, [t, t']) \in \mathcal{G} \), then
\[
\hat{\omega}\uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff } \text{prim}(E) \\
\hat{\omega}\uparrow(\omega\uparrow(\gamma)) & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

\[ \square \]

Definition 6.22
Immediate initiator occurrence: Let \( \gamma \triangleq \Gamma(E, [t, t']) \in \mathcal{G} \), then
\[
\alpha\uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff } \text{prim}(E) \\
\gamma' \text{ where } \gamma' \in \{ \gamma'' | \text{initiate}(\gamma'', \gamma) \} & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

\[ \square \]
Definition 6.23
Ultimate initiator occurrence: Let $\gamma \vdash \Gamma(E, [t,t']) \in \mathcal{S}$, then
$$\hat{\alpha} \uparrow(\gamma) = \begin{cases} 
\gamma & \text{iff } \text{prim}(E) \\
\hat{\alpha} \uparrow(\alpha \uparrow(\gamma)) & \text{iff } \neg \text{prim}(E)
\end{cases}$$

Definition 6.24
Precedes:
$$\gamma_1 \prec \gamma_2 \overset{\Delta}{=} \text{end}(\text{span}(\gamma_1)) < \text{end}(\text{span}(\gamma_2)) \lor \left( \text{end}(\text{span}(\gamma_1)) = \text{end}(\text{span}(\gamma_2)) \land \left( \left( \text{c}_\text{ord}(\text{type}(\gamma_1)) < \text{c}_\text{ord}(\text{type}(\gamma_2)) \lor \omega \uparrow(\gamma_1) < \omega \uparrow(\gamma_2) \right) \right) \right)$$

Definition 6.25
Succeeds:
$$\gamma_1 \succ \gamma_2 \overset{\Delta}{=} \gamma_2 \prec \gamma_1$$
Definition 6.26
Overlaps:

\[ \gamma_1 \parallel \gamma_2 \triangleq \neg (\text{end}(	ext{span}(\gamma_1)) < \text{start}(	ext{span}(\gamma_2)) \lor \text{start}(	ext{span}(\gamma_1)) > \text{end}(	ext{span}(\gamma_2))) \]

□

Definition 6.27
Expiration predicate:

\[ \text{expired}(\gamma) \triangleq \text{now} - \text{start}(	ext{span}(\gamma)) > \text{expire}(<\gamma>) \]

□

B.4 Solicitor Event Property Functions

Definition 6.28
All-immediate-contributing-types-of:

\[
\text{all}_{\text{operands}}(E) = \begin{cases} 
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \{E_1, E_2\} & \text{iff } (E = E_1; E_2) \lor (E = E_1 \lor E_2) \lor (E = E_1 \triangle E_2) \\
(c): \{E_1, E_T\} & \text{iff } (E = E_1 + T) \\
(d): \{E_1, E_2, E_3\} & \text{iff } (E = \text{N}(E_1, E_2, E_3)) \lor (E = A(E_1, E_2, E_3)) \lor (E = \text{CA}(E_1, E_2, E_3)) 
\end{cases}
\]

□
Definition 6.29
All-contributing-types-of:
\[
\text{all operands}(E) =
\begin{cases}
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \bigcup_{E' \in \text{all operands}(E)} \text{all operands}(E') & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

□

Definition 6.30
Immediate-contributing-initiator-types-of:
\[
\alpha(E) =
\begin{cases}
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \{E_1\} & \text{iff } (E = E_1 : E_2) \lor (E = N(E_1, E', E_2)) \lor (E = E_1 + T) \\
(c): \{E_1, E_2\} & \text{iff } (E = E_1 \lor E_2) \lor (E = E_1 \Delta E_2) \\
(d): \{E_2\} & \text{iff } E = A(E_1, E_2, E_3) \lor E = CA(E_1, E_2, E_3)
\end{cases}
\]

□

Definition 6.31
Contributing-initiator-types-of:
\[
\hat{\alpha}(E) =
\begin{cases}
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \bigcup_{E' \in \alpha(E)} \hat{\alpha}(E') & \text{iff } \neg \text{prim}(E)
\end{cases}
\]

□
B.5 Event Context Predicates

Definition 6.32
Immediate-contributing-terminator-types-of:

\[
\omega(E) = \begin{cases} 
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \{E_2\} & \text{iff } (E = E_1; E_2) \lor (E = N(E_1, E', E_2)) \\
(c): \{E_1, E_2\} & \text{iff } (E = E_1 \lor E_2) \lor (E = E_1 \Delta E_2) \\
(d): \{E_2\} & \text{iff } E = A(E_1, E_2, E_3) \lor E = CA(E_1, E_2, E_3)
\end{cases}
\]

Definition 6.33
Contributing-terminator-types-of:

\[
\hat{\omega}(E) = \begin{cases} 
(a): \{E\} & \text{iff } \text{prim}(E) \\
(b): \bigcup_{E' \in \omega(E)} \hat{\omega}(E') & \text{iff } \neg\text{prim}(E)
\end{cases}
\]

B.5 Event Context Predicates

The event context predicates for chronicle (Def. 6.39 on p. 129), recent (Def. 6.40 on p. 129), continuous (Def. 6.41 on p. 130), and, finally, general (Def. 6.42 on p. 130) event contexts defined in Ch. 6 are repeated here:

Definition 6.34
General form of context predicate:
Let \( X \) be the context predicate, then:

\[
X(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \overset{\Delta}{=} X_\alpha(E, \gamma_\alpha, \mathcal{E}_\alpha) \land X_\omega(E, \gamma_\omega, \mathcal{E}_\omega) \land X_{\alpha\omega}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega)
\]

□
Definition 6.35
All Non-expired Event Occurrences of a Set of Event Types:
\[
\text{all}_\text{of}_\text{types}(E) = \{ \gamma \mid \gamma \in G \wedge \text{type}(\gamma) \in E \wedge \neg \text{expired}(\gamma) \}
\]
□

Definition 6.36
Basic dynamic set definitions for context predicates:
\[
\begin{align*}
\text{ALL}_\text{a} = \text{all}_\text{of}_\text{types}(E_\text{a}) & \quad (\text{all event occurrences of initiator event types } E_\text{a}) \\
\text{ALL}_\text{w} = \text{all}_\text{of}_\text{types}(E_\text{w}) & \quad (\text{all event occurrences of terminator event types } E_\text{w})
\end{align*}
\]
□

Definition 6.37
Dynamic set definitions of event type \(E\) for context predicates:
\[
\begin{align*}
\text{COMP} & = \text{com}(U_\text{a}(E) \cup U_\text{w}(E)) \quad (\text{all composed event occurrences}) \\
\text{INVAL}_\text{a} & = I_\text{a}(E) \quad (\text{invalid as initiators to form composite event occurrences}) \\
\text{INVAL}_\text{w} & = I_\text{w}(E) \quad (\text{invalid as terminators to form comp. event occurrences}) \\
\text{USED}_\text{a} & = \text{iot}(U_\text{a}(E)) \quad (\text{used as initiators for composite event occurrences}) \\
\text{USED}_\text{w} & = \text{iot}(U_\text{w}(E)) \quad (\text{used as terminators for composite event occurrences})
\end{align*}
\]
□

Definition 6.38
Derived dynamic set definitions of event type \(E\) for context predicates:
\[
\begin{align*}
\text{CONS}_\text{a} & = (\text{USED}_\text{a} \cup \text{INVAL}_\text{a}) \quad (\text{consumed as initiators}) \\
\text{CONS}_\text{w} & = (\text{USED}_\text{w} \cup \text{INVAL}_\text{w}) \quad (\text{consumed as terminators}) \\
\text{UNCONS}_\text{a} & = (\text{ALL}_\text{a} \setminus (\text{CONS}_\text{a} \cup \text{CONS}_\text{w})) \quad (\text{unconsumed initiators}) \\
\text{UNCONS}_\text{w} & = (\text{ALL}_\text{w} \setminus (\text{CONS}_\text{a} \cup \text{CONS}_\text{w})) \quad (\text{unconsumed terminators}) \\
\text{VALID}_\text{a} & = (\text{ALL}_\text{a} \setminus \text{INVAL}_\text{a}) \quad (\text{initiators valid for composition}) \\
\text{VALID}_\text{w} & = (\text{ALL}_\text{w} \setminus \text{INVAL}_\text{w}) \quad (\text{terminators valid for composition})
\end{align*}
\]
□
Definition 6.39
Chronicle context predicate:

\[ \text{chronicle}_\alpha(E, \gamma, E) \overset{\Delta}{=} \forall \gamma \in \mathcal{UNCONS}_\alpha (\gamma \not\alpha \Rightarrow \gamma \prec \gamma) \land \gamma \in \mathcal{UNCONS}_\alpha \]

\[ \text{chronicle}_\omega(E, \gamma, E) \overset{\Delta}{=} \forall \gamma \in \mathcal{UNCONS}_\omega (\gamma \not\omega \Rightarrow \gamma \prec \gamma) \land \gamma \in \mathcal{UNCONS}_\omega \]

\[ \text{chronicle}_{\alpha\omega}(E, \gamma, \gamma, E, E) \overset{\Delta}{=} \text{true} \]

\[ \square \]

Definition 6.40
Recent context predicate:

\[ \text{recent}_\alpha(E, \gamma, E) \overset{\Delta}{=} \gamma \in \mathcal{VAICT}_\alpha \]

\[ \text{recent}_\omega(E, \gamma, E) \overset{\Delta}{=} \text{chronicle}_\omega(E, \gamma, E) \]

\[ \text{recent}_{\alpha\omega}(E, \gamma, \gamma, \gamma, E, E) \overset{\Delta}{=} \neg \exists \gamma \in \mathcal{VAICT}_\omega (\gamma \not\alpha \land \gamma \prec \gamma \prec \gamma \omega) \]

\[ \square \]

Definition 6.41
Continuous context predicate:

\[ \text{continuous}_\alpha(E, \gamma, E) \overset{\Delta}{=} \text{chronicle}_\alpha(E, \gamma, E) \]

\[ \text{continuous}_\omega(E, \gamma, E) \overset{\Delta}{=} \gamma \in \mathcal{VAICT}_\omega \]

\[ \text{continuous}_{\alpha\omega}(E, \gamma, \gamma, \gamma, E, E) \overset{\Delta}{=} \neg \exists \gamma \in \mathcal{VAICT}_\omega (\gamma \not\omega \land \gamma \prec \gamma \prec \gamma \omega) \]

\[ \square \]
Definition 6.42
General event context:

\[ \text{general } \alpha (E, \gamma_\alpha) \equiv \gamma_\alpha \in \mathcal{V}_{\text{ALID}_\alpha} \]

\[ \text{general } \omega (E, \gamma_\omega) \equiv \gamma_\omega \in \mathcal{V}_{\text{ALID}_\omega} \]

\[ \text{general } \alpha \omega (E, \gamma_\alpha, \gamma_\omega) \equiv \text{gen combination not used before}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \land \\
\quad (\mathcal{E}_\alpha \neq \mathcal{E}_\omega \Rightarrow \text{all prec occs processed}_1(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega)) \land \\
\quad (\mathcal{E}_\alpha = \mathcal{E}_\omega \Rightarrow \text{all prec occs processed}_2(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega)) \]

\[ \text{all prec occs processed}_1(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \equiv \\
\forall \gamma, \gamma' (\text{type}(\gamma) = E \land \text{initiate}(\gamma, \gamma') \land \gamma \succ \gamma') \Rightarrow \\
\quad \forall \gamma'' (\gamma'' \in \mathcal{V}_{\text{ALID}_\omega} \land (\gamma' \prec \gamma'' \lor \gamma' = \gamma'') \Rightarrow \\
\quad \exists \gamma''' \text{type}(\gamma'''') = E \land \\
\quad \text{initiate}(\gamma', \gamma''') \land \text{terminate}(\gamma'', \gamma'''))) \land \\
\forall \gamma \in \mathcal{V}_{\text{ALID}_\omega} \land (\gamma \prec \gamma \lor \gamma = \gamma) \land \gamma \succ \gamma \Rightarrow \\
\exists \gamma' \text{terminate}(\gamma', \gamma') \]

\[ \text{all prec occs processed}_2(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \equiv \\
\forall \gamma, \gamma' (\gamma \in \mathcal{V}_{\text{ALID}_\alpha} \land \gamma \prec \gamma' \Rightarrow \exists \gamma'' \text{initiate}(\gamma, \gamma'')) \land \\
\forall \gamma, \gamma' (\gamma \in \mathcal{V}_{\text{ALID}_\omega} \land \gamma \succ \gamma \Rightarrow \exists \gamma'' \text{terminate}(\gamma, \gamma'')) \]

\[ \text{gen combination not used before}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \equiv \\
\forall \gamma, \gamma', \gamma'' (\text{type}(\gamma) = E \land \text{initiate}(\gamma', \gamma) \land \text{terminate}(\gamma'', \gamma) \land \gamma' \neq \gamma'' \Rightarrow \\
\quad \neg (\gamma_\alpha = \gamma' \land \gamma_\omega = \gamma'') \land \neg (\gamma_\alpha = \gamma'' \land \gamma_\omega = \gamma'))) \]
B.6 Event Operator Rules

Definition 6.44
Sequence:
Let $E = E_1; E_2$ and $\gamma \equiv \Gamma(E, [\text{start}(\text{span}(\gamma_1)), \text{end}(\text{span}(\gamma_2))])$ in the $s$-rule:

\[
\forall E \forall \gamma_1, \gamma_2 \in G : \\
type(\gamma_1) = E_1 \land type(\gamma_2) = E_2 \\
X(E, \gamma_1, \gamma_2, \{E_1\}, \{E_2\}) \\
\gamma_1 \ll \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2) \\
G \cup \{\gamma\} \cup (\{\gamma_1\} \cup (\{\gamma_2\} \\ s

\forall E \forall \gamma_2 \in G : \\
type(\gamma_2) = E_2 \\
X_\omega(E, \gamma_2, \{E_2\}) \\
\forall \gamma_1 \in G \ (type(\gamma_1) = E_1 \land X(E, \gamma_1, \gamma_2, \{E_1\}, \{E_2\}) \Rightarrow \gamma_1 \ll \gamma_2 \lor \gamma_1 \parallel \gamma_2) \\ s'

I_\omega(E) \cup \{\gamma_2\}

□

Proof of correctness of rules can be found in Table 6.1 on p. 136.

Definition 6.45
Conjunction:
Let $E = E_1 \triangle E_2$ in the following rules and let

$\gamma \equiv \Gamma(E, [\text{start}(\text{span}(\gamma_1)), \text{end}(\text{span}(\gamma_2))])$ in the $c[f]$-rule:

\[
\forall E \forall \gamma_1, \gamma_2 \in G : \\
type(\gamma_1) = E_1 \land type(\gamma_2) = E_2 \\
X(E, \gamma_1, \gamma_2, \{E_1\}, \{E_2\}) \\
\gamma_1 \ll \gamma_2 \\
G \cup \{\gamma\} \cup (\{\gamma_1\} \cup (\{\gamma_2\} \\ c[f]
Let $\gamma \overset{\lambda}{=} \Gamma(E, [\text{start(span}(\gamma_2), \text{end(span}(\gamma_1))])$ in the $c[r]$-rule:

$$\forall E \forall \gamma_1, \gamma_2 \in \mathcal{G} :$$

$$\text{type}(\gamma_1) = E_1 \land \text{type}(\gamma_2) = E_2$$

$$X(E, \gamma_2, \gamma_1, \{E_2\}, \{E_1\})$$

$$\gamma_2 \triangleright \gamma_1$$

$$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(E) \cup = \{(\gamma_2, \gamma)\} \quad \mathcal{U}_u(E) \cup = \{(\gamma_1, \gamma)\}$$

$c[r]$  

\[ \square \]

Proof of correctness of rules can be found in Table 6.3 on p. 139.

**Definition 6.46**

**Disjunction:**

Let $E = E_1 \varve E_2$ and let $\gamma \overset{\lambda}{=} \Gamma(E_1 \varve E_2, [\text{start(span}(\gamma_1), \text{end(span}(\gamma_1))])$ in the $d[l]$-rule:

$$\forall E \forall \gamma_1 \in \mathcal{G} :$$

$$\text{type}(\gamma_1) = E_1$$

$$X(E, \gamma_1, \gamma_1, \{E_1, E_2\}, \{E_1, E_2\})$$

$$\text{true}$$

$$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(E) \cup = \{(\gamma_1, \gamma)\} \quad \mathcal{U}_u(E) \cup = \{(\gamma_1, \gamma)\}$$

$d[l]$  

Let $\gamma \overset{\lambda}{=} \Gamma(E, [\text{start(span}(\gamma_2), \text{end(span}(\gamma_2))])$ in the $d[r]$-rule:

$$\forall E \forall \gamma_1 \in \mathcal{G} :$$

$$\text{type}(\gamma_1) = E_2$$

$$X(E, \gamma_1, \gamma_1, \{E_1, E_2\}, \{E_1, E_2\})$$

$$\text{true}$$

$$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(E) \cup = \{(\gamma_1, \gamma)\} \quad \mathcal{U}_u(E) \cup = \{(\gamma_1, \gamma)\}$$

$d[r]$  

\[ \square \]

Proof of correctness of rules can be found in Table 6.4 on p. 140.
Definition B.2

Non-occurrence: Let event expression be $E = N(E_1, E_2, E_3)$ and $\gamma = \Gamma(E, [\text{start}(\text{span}(\gamma_1)), \text{end}(\text{span}(\gamma_3))])$ in the $n$-rule:

\[
\forall E \forall \gamma_1, \gamma_3 \in G : \\
type(\gamma_1) = E_1 \land type(\gamma_3) = E_3 \\
X(E, \gamma_1, \gamma_3, \{E_1\}, \{E_3\}) \\
\gamma_1 \prec \gamma_3 \land \lnot(\gamma_1 \parallel \gamma_3) \\
\exists \gamma_2 \in G \land (type(\gamma_2) = E_2 \Rightarrow \gamma_1 \prec \gamma_2 \land \lnot(\gamma_1 \parallel \gamma_2) \land \gamma_2 \prec \gamma_3 \land \lnot(\gamma_2 \parallel \gamma_3)) \\
G \cup = \{\gamma\} \quad U_a(E) \cup = \{\langle \gamma_1, \gamma \rangle\} \quad U_c(E) \cup = \{\langle \gamma_3, \gamma \rangle\} \\
\]

For the $n$-rule, for each pair of event occurrences of $E_1$ and $E_3$ without an intermediate occurrence of $E_2$, compose a not event occurrence denoting the non-occurrence of an $E_2$ event in the interval of the pair.

For the $n''$-rule, invalidate an $E_1$ occurrence that cannot be an initiator of a not event. The proof of the operator rules of not can be found in Table B.1 on the following page.
### Table B.1: Proof table for the non-occurrence operator

<table>
<thead>
<tr>
<th>#</th>
<th>Precedence</th>
<th>Overlap</th>
<th>Rule</th>
<th>U/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>(1, -3, 2)</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>(1, -3, 2)</td>
<td>F</td>
<td>?</td>
<td>n'</td>
</tr>
<tr>
<td>4.</td>
<td>(1, -2, 3)</td>
<td>?</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>5.</td>
<td>(1, -2, 3)</td>
<td>?</td>
<td>T</td>
<td>n'</td>
</tr>
<tr>
<td>6.</td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>(2, -3, 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>(2, -1, 3)</td>
<td>?</td>
<td>?</td>
<td>n'</td>
</tr>
<tr>
<td>9.</td>
<td>(3, ?, ?)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Definition B.3

**Aperiodic:** Let $E = A(E_1, E_2, E_3)$ and $\gamma = \Gamma(E, \text{span}(\gamma_2))$ in the $a$-rule:

\[
\forall E \forall \gamma_2 \in \mathcal{G} : \\
\text{type}(\gamma_2) = E_2 \\
X(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\}) \\
\exists \gamma_1 \in \mathcal{G} (\gamma_1 \prec \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2) \land \\
\neg \exists \gamma_3 \in \mathcal{G} (\text{type}(\gamma_3) = E_3 \land \gamma_3 \succ \gamma_1 \land \neg(\gamma_3 \parallel \gamma_1) \land \gamma_3 \prec \gamma_2)) \\
\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(E) \cup = \{(\gamma_2, \gamma)\} \quad \mathcal{U}_\omega(E) \cup = \{(\gamma_2, \gamma)\} \quad a
\]

\[
\forall E \in \mathcal{E}_m \forall \gamma_2 \in \mathcal{G} : \\
\text{type}(\gamma_2) = E_2 \\
X(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\}) \\
\forall \gamma_1 \in \mathcal{G} (\gamma_1 \prec \gamma_2 \land \neg(\gamma_1 \parallel \gamma_2) \Rightarrow \\
\exists \gamma_3 \in \mathcal{G} (\text{type}(\gamma_3) = E_3 \land \gamma_3 \succ \gamma_1 \land \neg(\gamma_3 \parallel \gamma_1) \land \gamma_3 \prec \gamma_2)) \\
\mathcal{I}_a(E) \cup = \{(\gamma_2, \gamma)\} \quad \mathcal{I}_\omega(E) \cup = \{(\gamma_2, \gamma)\} \quad a'
\]
### B.6 Event Operator Rules

<table>
<thead>
<tr>
<th>#</th>
<th>Precedence</th>
<th>Overlap</th>
<th>Rule</th>
<th>U/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(1)</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>(1,−3, 2)</td>
<td>T</td>
<td>a′</td>
<td>Iα, Iω</td>
</tr>
<tr>
<td>3.</td>
<td>(1,−3, 2)</td>
<td>F</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>4.</td>
<td>(1,−2, 3)</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>5.</td>
<td>(2)</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6.</td>
<td>(2,−3, 1)</td>
<td></td>
<td></td>
<td>a′</td>
</tr>
<tr>
<td>8.</td>
<td>(2,−1, 3)</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>9.</td>
<td>(3,?,?)</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Table B.2: Proof table for the aperiodic operator

**Definition B.4**

**Relative temporal:** Let \( E = E_1 + T, \ t = \text{end}(\text{span}(\gamma_1) + T \) and \( \gamma = \Gamma(E, [\text{start}(\text{span}(\gamma_1)), t]) \) in the \( r \)-rule:

\[
\forall E \forall \gamma_1 \in \mathcal{G} : \\
\text{type}(\gamma_1) = E_1 \\
X(E, \gamma_1, \Gamma(E_T, [t,t]), \{E_1\}, \{E_T\}) \\
t \leq \text{now} \\
\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(\mathcal{E}) \cup = \{\langle \gamma_1, \gamma \rangle\} \quad \mathcal{U}_\omega(\mathcal{E}) \cup = \{\langle \Gamma(E_T, [t,t]), \gamma\rangle\} \\
\begin{array}{l}
\end{array}
\]

The proof for the relative temporal operator is trivial and is left out.

**Definition 8.1**

**Chronicle aperiodic:** Let \( \text{UNCONS}^A_\alpha \triangleq \text{ALL}_\alpha \setminus \text{I}^A_\alpha(E) \), \( \text{UNCONS}^A_\omega \triangleq \text{ALL}_\omega \setminus \text{I}^A_\omega(E) \) (where \( \text{ALL}_\alpha \) and \( \text{ALL}_\omega \) are defined in Def. 6.36 on p. 128, and \( \text{UNCONS}^A_\alpha \) and \( \text{UNCONS}^A_\omega \) are similar to \( \text{UNCONS}_\alpha \) and \( \text{UNCONS}_\omega \) in Def. 6.38 on p. 128) in:

\[
\text{ca}_\alpha(\mathcal{E}_\alpha) \Delta \\
\forall \gamma \in \text{UNCONS}^A_\alpha (\gamma_\alpha \neq \gamma \Rightarrow \gamma_\alpha \prec \gamma) \land \gamma_\alpha \in \text{UNCONS}^A_\alpha \\
\text{ca}_\omega(\mathcal{E}_\omega) \Delta \\
\forall \gamma \in \text{UNCONS}^A_\omega (\gamma_\omega \neq \gamma \Rightarrow \gamma_\omega \prec \gamma) \land \gamma_\omega \in \text{UNCONS}^A_\omega 
\]
Let $E \equiv CA(E_1, E_2, E_3)$ and $\gamma \equiv \Gamma(E, \text{span}(\gamma_2))$ in the $ca$-rules:

$$\forall E \forall \gamma_1, \gamma_2 \in G :$$

$$\text{type}(\gamma_1) = E_1 \land \text{ca}_{ei}(E, \gamma_1, \{E_1\}) \land \text{type}(\gamma_2) = E_2$$

$$X(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\})$$

$$\gamma_1 \prec \gamma_2 \land \neg (\gamma_1 \parallel \gamma_2)$$

$$\neg \exists \gamma_3 \in G (\text{type}(\gamma_3) = E_3 \land \gamma_3 \not\in \text{I}_a^A(E) \land \gamma_1 \prec \gamma_3 \land \neg (\gamma_1 \parallel \gamma_3) \land \gamma_2 \parallel \gamma_3)$$

$$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(E) \cup = \{(\gamma_2, \gamma)\} \quad \mathcal{U}_\omega(E) \cup = \{(\gamma_2, \gamma)\}$$

$$\forall E \forall \gamma_2 \in G :$$

$$\text{type}(\gamma_2) = E_2$$

$$X_\omega(E, \gamma_2, \{E_2\})$$

$$\forall \gamma_1 \in G (\text{type}(\gamma_1) = E_1 \land \text{ca}_{ei}(E, \gamma_1, \{E_1\}) \land$$

$$X(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\}) \Rightarrow$$

$$\gamma_2 \prec \gamma_1 \lor \gamma_2 \parallel \gamma_1)$$

$$\mathcal{T}_a(E) \cup = \{\gamma_2\} \quad \mathcal{T}_\omega(E) \cup = \{\gamma_2\}$$

$$\forall E \forall \gamma_1, \gamma_3 \in G :$$

$$\text{type}(\gamma_1) = E_1 \land \text{type}(\gamma_3) = E_3$$

$$\text{ca}_{ei}(E, \gamma_1, \{E_1\}) \land \text{ca}_{ei}(E, \gamma_3, \{E_3\})$$

$$\gamma_1 \prec \gamma_3 \land \neg (\gamma_1 \parallel \gamma_3)$$

$$\neg \exists \gamma_2 \in G (\text{type}(\gamma_2) = E_2 \land X(E, \gamma_2, \gamma_2, \{E_2\}, \{E_2\}) \land$$

$$\gamma_1 \prec \gamma_2 \land \neg (\gamma_1 \parallel \gamma_2) \land \gamma_2 \parallel \gamma_3)$$

$$\mathcal{T}_a^A(E) \cup = \{\gamma_1\} \quad \mathcal{T}_\omega^A(E) \cup = \{\gamma_3\}$$

$$\forall E \forall \gamma_3 \in G :$$

$$\text{type}(\gamma_3) = E_3$$

$$\text{ca}_{ei}(E, \gamma_3, \{E_3\})$$

$$\forall \gamma_1 \in G (\text{type}(\gamma_1) = E_1 \land \text{ca}_{ei}(E, \gamma_1, \{E_1\}) \Rightarrow$$

$$\gamma_3 \prec \gamma_1 \lor \gamma_3 \parallel \gamma_1)$$

$$\mathcal{T}_a^A(E) \cup = \{\gamma_3\}$$
The axioms of the programming logic are:

**Axiom C.1**

**Skip:** \( \{ P \} \text{skip}\{ P \} \)

**Axiom C.2**

**Assignment:** \( \{ P^x \} x := e \{ P \} \)

**Axiom C.3**

**Swap:** \( \{ P^x_{v1,v2} \} v1 := v2 \{ P \} \)

The inference rules of programming logic are:
\( P' \Rightarrow P, \{P\} S\{Q\}, Q \Rightarrow Q' \)

\( \{P'\} S\{Q'\} \)

Consequence

\( \{P\} S_1\{Q\}, \{Q\} S_2\{R\} \)

\( \{P\} S_1; S_2\{R\} \)

Composition

\( P \land \neg (B_1 \lor \ldots \lor B_n) \Rightarrow Q \)

\( \{P \land B_i\} S_i\{Q\}, 1 \leq i \leq n \)

\( \{P\} IF\{Q\} \)

Alternative

\( \{I \land B_i\} S_i\{I\}, 1 \leq i \leq n \)

\( \{I\} DO\{I \land \neg (B_1 \lor \ldots \lor B_n)\} \)

Iterative
This appendix contains graphs depicting the distribution of response times of executing experiments in Section 12.2.1 on p. 268.
Figure D.1: Distribution of response times for chronicle context, $|E|=1$, $|G|=10$

Figure D.2: Distribution of response times for chronicle context, $|E|=1$, $|G|=100$
Figure D.3: Distribution of response times for chronicle context, $|E|=1$, $|G|=1000$.

Figure D.4: Distribution of response times for chronicle context, $|E|=1$, $|G|=10000$. 
Figure D.5: Distribution of response times for chronicle context, $|E| = 10$, $|G| = 10$

Figure D.6: Distribution of response times for chronicle context, $|E| = 10$, $|G| = 100$
Figure D.7: Distribution of response times for chronicle context, $|E| = 10$, $|G| = 1000$.

Figure D.8: Distribution of response times for chronicle context, $|E| = 10$, $|G| = 10000$. 
Distribution of Response Times

Figure D.9: Distribution of response times for chronicle context, $|E| = 50$, $|G| = 10$

Figure D.10: Distribution of response times for chronicle context, $|E| = 50$, $|G| = 100$
Figure D.11: Distribution of response times for chronicle context, $|E| = 50$, $|G| = 1000$

Figure D.12: Distribution of response times for recent context, $|E| = 1$, $|G| = 10$
Figure D.13: Distribution of response times for recent context, $|E| = 1$, $|G| = 100$

Figure D.14: Distribution of response times for recent context, $|E| = 1$, $|G| = 1000$
Figure D.15: Distribution of response times for recent context, $|E| = 1$, $|G| = 10000$

Figure D.16: Distribution of response times for recent context, $|E| = 10$, $|G| = 10$
Figure D.17: Distribution of response times for recent context, $|E|=10$, $|G|=100$

Figure D.18: Distribution of response times for recent context, $|E|=10$, $|G|=1000$
Figure D.19: Distribution of response times for recent context, $|E|=10$, $|G|=10000$

Figure D.20: Distribution of response times for recent context, $|E|=50$, $|G|=10$
Figure D.21: Distribution of response times for recent context, $|E|=50$, $|G|=100$

Figure D.22: Distribution of response times for recent context, $|E|=50$, $|G|=1000$
Appendix E

Recent and Chronicle Dependent on Size of Type
Recent and Chronicle Dependent on Size of Type

Figure E.1: Response times for recent context, $|G|=1000$

Figure E.2: Response times for recent context, $|G|=5000$
Figure E.3: Response times for recent context, $|\mathcal{G}|=10000$

Figure E.4: Response times for chronicle context, $|\mathcal{G}|=1000$
Figure E.5: Response times for chronicle context, $|G|=5000$

Figure E.6: Response times for chronicle context, $|G|=10000$


No 503  **Johan Ringström**: Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.


<table>
<thead>
<tr>
<th>No</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
</table>


