Event Composition Model:

Achieving Naturalness In Runtime Enforcement

Somayeh Malakuti Khah Olun Abadi

Event Composition Model:

Achieving Naturalness in Runtime Enforcement

Somayeh Malakuti Khah Olun Abadi

Ph.D. dissertation committee:

Chairman and secretary:

Prof. Dr. Ir. A.J. Mouthaan, University of Twente, The Netherlands

Promoter:

Prof. Dr. Ir. Mehmet Akşit, University of Twente, The Netherlands

Assistant promoter:

Dr. Christoph Bockisch, University of Twente, The Netherlands

Members:

Prof. Dr. Jaco van de Pol, University of Twente, The Netherlands
Prof. Dr. Jozef Hooman, Radboud University, The Netherlands
Prof. Dr. Wouter Joosen, Katholieke Universiteit Leuven, Belgium
Dr. Grigore Rosu, University of Illinois at Urbana-Champaign, The United States
Dr. Oleg Sokolsky, University of Pennsylvania, The United States
Dr. Ir. J. Broenink, University of Twente, The Netherlands

CTIT Ph.D. thesis series no. 11-205. Center for Telematics and Information Technology (CTIT), P.O. Box 217 - 7500 AE Enschede, The Netherlands.

This work has been partially carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Government under the Bsik program. The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 978-90-365-3246-4
ISSN 1381-3617 (CTIT Ph.D. thesis series no. 11-205)
DOI: 10.3990/1.9789036532464
IPA Dissertation Series 2011-14

Cover design by Ram Kottumakulal and Javad Malakuti
Printed by PrintPartners Ipskamp, Enschede, The Netherlands

Event Composition Model:

Achieving Naturalness in Runtime Enforcement

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof. Dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday the 15th of September 2011 at 12.45

by

Somayeh Malakuti Khah Olun Abadi

born on the 23rd of September 1980
in Tehran, Iran

This dissertation is approved by

Prof. Dr. Ir. Mehmet Akşit (promoter)
Dr. Christoph Bockisch (assistant promoter)

To my parents
Hassan and Nazanin

Acknowledgements

Life is a composition of events, a composition that is guided by our dreams. No matter how big and unreachable our dreams seem to be, the events of our life are composed in such a way that they come true, with the help of several people. Studying till Ph.D. level was an old dream of mine. During this journey I encountered several people who played a significant role to convert my dream to reality. I would like to thank all these people at this very important stage of my life.

In the first place, I must thank Prof. Mehmet Aksit who converted my dream to reality by offering me a Ph.D. position and by accompanying me in every step of this long journey. Our frequent meetings in which Mehmet was constantly inspiring me and was sharing his knowledge with me, our long paper-writing meetings which we used to have our dinner at McDonald's and until 2 or 3 o'clock in the morning Mehmet was patiently teaching me that there is a huge difference between writing poems and technical papers, our midnight meetings about the core concepts of the thesis, our long thesis-writing nights when we were correcting the thesis in Starbucks, our conference trips where we were writing papers in taxi, airport and airplane, the opportunities that Mehmet created for me to meet excellent researchers all around the world, our frequent discussions about Rumi and his poems, all and all fill my eyes with the tears of gratitude. I am also very thankful to the family of Mehmet for all the attention that they gave me during last four years.

In the first year of my Ph.D., I had the chance to work with Dr. Bedir Tekinerdogan as my daily supervisor. I appreciate his contribution to my research, and I wish him the best of success in his life in Turkey. In the end of my second year, Dr. Christoph Bockisch joined the group and kindly offered his help to us. His concrete view on the subjects played an important role in achieving and publishing concrete results. I am also very thankful to him for this effort.

I learned about runtime enforcement and its application in TRADER during the time I was closely working with Prof. Jozef Hooman. I thank Prof. Hooman for his effort in my research, and for being a member of my committee. I would also like to thank the other members of the TRADER project, in particular David Watts and Roland Mathijssen for their valuable feedbacks in our project meetings.

I would also like to thank the other members of my committee: Prof. Wouter Joosen, Prof. Jaco van de Pol, Dr. Jan Broenink, Dr. Oleg Sokolsky and Dr. Grigore Rosu. It is an honor for me that you dedicated your time to read my thesis, and to travel long distances to participate in my defense. Your feedbacks improved the quality of my thesis.

I am very thankful to the members of the TRESE group, whose loving, caring and cheerful attitude eased my job to get adapted to the new country and the new culture. In particular, I would like to thank Hasan (Super Hasan) and Ismenia for helping me through the formal procedures, and in finding my way in the Netherlands. I am very thankful to Eduardo for helping me in the final arrangements for the thesis. I thank Lodewijk and his Compose* team, whose work inspired me a lot. I am very grateful to our secretaries during last four years, Ellen, Joke, Hilda, Nathalie and Jeanette for their valuable administrative support. I appreciate all the love and attention that Jeanette gave me during tough times.

I am very grateful to Prof. Shmuel Katz for the inspiring topics that he brought up during our meetings. During his short visit from TRESE, we published two papers in highly prestigious conferences, which were also inspiring for the concepts of this thesis. I would also like to thank Dr. Oleg Sokolsky and Dr. Klaus Havelund for giving me the opportunity to attend Dagstuhl seminar on Runtime Verification, Diagnosis, Planning and Control for Autonomous Systems. It was an inspiring event and motivated me to continue my research in the field of runtime verification. Shortly before finishing the thesis, I had a chance to attend the AOM-ReMoDD workshop in Barbados. The productive discussions in the workshop inspired me for the presentation of this thesis. I thank Dr. Jorg kienzle, Prof. Robert France, Dr. Ana Moreira, and the other participants of the workshop for this matter.

When moving to another country, we must put all of our beloved ones behind, with a great hope that we may meet other caring, loving and reliable people. I consider myself very lucky that I could meet such people during my stay in the Netherlands. Along this line, I would like to send my very special thanks to Nima, my first Iranian friend in the Netherlands, and his girlfriend Katja for their constant unconditional support and attention during last four years. I am convinced that it would be much more difficult to settle my life in the Netherlands without their help.

I am very thankful to Ram and Saba for being true friends in tough times, when I needed the pain to be lifted from my heart by people who could truly feel the pain in their heart. I wish I could find a way to express my gratitude to you both for receiving me in your house, in the last week of the thesis-writing period when the stress had blocked my progress. I also thank Ram for giving his artistic touch to the cover of my thesis.

The friendship with Amirmehdi, Matin, Faiza, Faizan, little Farzan, Esly, Hadi, Narges, Andre, Robin, and many others created a lot of unforgettable moments for me. I hope our friendship lasts, regardless of our geographical location.

Although I had to leave my lovely friends and move to the Netherlands, they taught me that a true friendship is never influenced by geographical distances. My dear Fereshteh, when I met you for the first time, I never imagined that you will be *THE* friend who would accompany me along this way. Thanks a lot for always being there for me, listening to me, holding my hand, laughing with me, crying with me, and for all the love and attention that you gave me during our 16 years of friendship. Your personality truly justifies the meaning of your name, i.e. an angel.

I never consider it a coincident that in the beginning of my career as a software engineer I had Mr. Yusef Mehrdad as my technical manager. Dear Yusef, there are many many things that I must thank you for. I thank you for watching every single step of mine to make sure that I am in the right path; for teaching me that to discover more, our heart must accompany our brain, for believing in me more than I did; for pushing me toward my dreams; and for all the good time that you, your lovely wife Roya, and your family Mohammad and Raana make me when I visit Iran.

Dear Narges, you taught me that if friends want to support each other, nothing, even the long distance between Australia and the Netherlands, can prevent them. I think if something happened to me in the Netherlands, you would be the first one to discover it, even before my friends in the Netherlands. Thanks a lot for all the love and attention that you gave me during last four years.

There are many other friends whose presence always encourages me to travel to Iran. I specifically thank my dear friend Ali Abdolrahmani who has always been a great source of inspiration for me. I am also very thankful to Nasrin, Reza, Azadeh, Hamed, and my cousin Zahra for all their remote support. I send very special thanks to Roxana, whose magical words easily took me out of dark and tough times. Although we met for a short time, I have learnt a lot from her and I still feel the influence of her words in my life. I also would like to thank Dr. Saeed Jalili, the supervisor of my Master assignment, for approving my thesis to be in

the field of aspect-orientation, and for all the support that he provided during my assignment.

I am grateful to my brothers Nasser, Yasser, Javad and to my sister-in-law Fatemeh, for taking good care of our parents and for making very good time for me when I visit Iran. After being away from you, I realize your significant role in my life. In particular, I thank Nasser for perfectly playing the role of a big brother, who always assures me that I should not be worried about our family because he has everything under control. Fatemeh, thanks a lot for becoming the sister that I did not have. I know how hard it can be to keep a balance between emotion and logic in a family whose majority of members are males. I wish I was there to give you a hand.

Yasser, our short age difference makes us emotionally very close to each other. Thank you for all the love and attention that you always give me, for constantly boosting my self-confidence, and for openly listening to my feelings and guiding me during hard times.

Javad thank you for making me feel as a big sister, for all the motivations that you give me, for all the jokes that you make to cheer me up, and for giving your artistic touch to the cover of my thesis. I appreciate your effort to occupy my place in mom's heart, which does not seem to be as easy as occupying my room in the house.

I dedicate a very special thank to my parents, for always believing in me and doing whatever they could to make this happen. I deeply thank them for being a motivation for my progress, rather than a barrier; for constantly reminding me that "easy days will come soon if I am patient enough"; for always assuring me that everything is all right with them, although I knew they were going through all those tough times without me being there for them. *This thesis was a team work, you were the most significant members of the team, and you always will be.*

Somayeh Malakuti
September 2011

Abstract

Runtime enforcement techniques are introduced in the literature to cope with the failures that occur while software is being executed in its target environment. These techniques may also offer diagnosis and recovery actions to respectively identify the causes of the failures and to heal them.

Since the development of runtime enforcement techniques can be complex, error-prone and costly, runtime enforcement frameworks are proposed to ease the development process. To this aim, these frameworks support various languages to specify the desired properties of software, to express the necessary diagnosis rules for detecting the causes of failures, and to define the recovery strategies. Based on the specifications, runtime enforcement frameworks generate code and integrate it with the software to be verified and/or healed. The code is usually generated in the same language that is used to implement the software, or in an intermediate language that abstracts the software.

Unfortunately, the specification languages employ the elements of the programming languages of the generated code, and therefore, they fall short in representing the runtime enforcement concepts *naturally*. By the term concept we mean a fundamental abstraction or definition that exists in most runtime enforcement techniques. As a result, implementation of runtime enforcement concepts may suffer from scattering and tangling. This reduces the modularity and compose-ability of the specifications of runtime enforcement concepts. Moreover, adoption of the elements of the underlying languages in specifications causes the specifications to become too specific to the employed programming languages and platforms. This reduces the reusability and comprehensibility of the specifications, and increases their complexity.

To facilitate a natural representation of runtime enforcement concepts, this thesis introduces a computation model termed as Event Composition Model, which respects the characteristic features of the runtime enforcement concepts. This computation

model offers a set of novel linguistic abstractions, called **events**, **event modules**, **reactors**, **reactor chains**, **event composition language** and **event constraint language**. Events represent changes in the states of interest. Event modules are means to group events, have input-output interfaces, and implementations. Reactors are the implementations of event modules. Reactor chains are groups of related reactors that process events in a sequence. The event composition language facilitates selecting the events of interest; and the event constraint language facilitates defining constraints among reactors or event modules.

An important issue is how to implement Event Composition Model effectively, by using current programming languages. For this purpose, the thesis evaluates the relevant programming languages with respect to their support for implementing Event Composition Model. The evaluation reveals that none of the existing languages can implement the model such that the desired quality attributes, such as modularity, abstractness and compose-ability are accomplished. Nevertheless, aspect-oriented languages offer promising features for this matter.

The thesis introduces the EventReactor language, as the successor of the Compose* aspect-oriented language, that implements Event Composition Model. The language is open-ended for new sorts of events and reactor types. This helps to represent new sorts of concepts. It makes use of the Prolog language as its event composition language. Reactors and reactor chains are parameterizable, and are defined separately from event modules. This increases the reusability of event modules and their implementations.

The systems of today are more and more implemented using multiple languages and this trend seems to continue also in the near future. The current runtime enforcement frameworks, unfortunately, fall short in supporting software implemented in different programming languages. In Event Composition Model, the event composition language facilitates selecting events from systems implemented in various languages. In the EventReactor language, the specifications are defined independently from any programming language, and the compiler of EventReactor facilitates generating code for Java, C and .Net languages. As a result, the specifications can be reused for software developed in different languages.

It is now more and more usual that applications are designed to run on distributed system architectures. Unfortunately, most runtime enforcement frameworks cannot be utilized for distributed systems. There are a few runtime enforcement frameworks that can work with distributed architectures. These systems, however, adopt specifications that contain information about the underlying process structure. This increases the complexity and reduces the reusability of the specifications if the process structure of software changes. The EventReactor language addresses this problem with the support of distribution transparency.

There are two basic ways in utilizing the EventReactor language: a) as an underlying language for the specification languages of runtime enforcement frameworks; b) as an implementation language of runtime enforcement techniques.

This thesis explains Event Composition Model, its implementation the EventReactor language, and the compiler of this language with the use of illustrative examples. The thesis makes use of an example runtime enforcement technique called Recoverable Process to evaluate the suitability of the EventReactor language in representing the concepts of interest naturally.

Contents

1 Introduction **1**

1.1 Motivations 2

1.2 Contributions 4

1.2.1 Identification of Problems in Implementing Runtime Enforcement Techniques 4

1.2.2 Enhancing Naturalness of Runtime Enforcement Concepts 5

1.2.3 The EventReactor Language: an Implementation of Event Composition Model 6

1.2.4 Supporting Multiple-Language Software 7

1.2.5 Supporting Multiple-Process Software 8

1.3 Thesis Overview 8

2 Requirements for Runtime Enforcement Frameworks **11**

2.1 A Canonical Model for Runtime Enforcement Frameworks 13

2.1.1 The Specification Layer 13

2.1.2 The Implementation Layer 15

2.2 Problem Statement 16

2.3 Requirements for Runtime Enforcement Frameworks 19
2.4 An Overview of Runtime Enforcement Frameworks 20
2.4.1 Dedicated Runtime Enforcement Frameworks 20
2.4.2 Languages Supporting Embedded Contracts 26
2.5 Summary 29

3 Enhancing Naturalness of Runtime Enforcement Concepts 31
3.1 Towards Natural Representation of Runtime Enforcement Concepts . 31
3.2 Event Composition Model 33
3.2.1 The Abstractions in Event Composition Model 33
3.2.2 The Meta-Model of Event Composition Model 35
3.3 Motivations for Adopting Event Composition Model 39
3.4 Implementing Event Composition Model in Object-Oriented Languages 40
3.5 Implementing Event Composition Model in Aspect-Oriented Languages 42
3.5.1 AspectJ 43
3.5.2 AspectJ Extensions 44
3.5.3 Compose* 45
3.5.4 AWED 46
3.5.5 AspectWerkz and EOS 47
3.6 Implementing Event Composition Model in Languages Supporting Implicit Invocation Mechanisms 48
3.6.1 C# 48
3.6.2 EventJava 49
3.6.3 EScala 49
3.6.4 Ptolemy 50
3.7 Summary of the Evaluation 51

3.8 Conclusion 52

4 EventReactor: an Implementation of Event Composition Model 55

4.1 The EventReactor Language 56

4.1.1 Specification of Events 56

4.1.2 Specification of Event Modules 58

4.1.3 Specification of Reactor Chains 60

4.1.4 Specification of Reactor Types 61

4.2 The Compiler of the EventReactor Language 62

4.2.1 Input and Output of the Compiler 64

4.2.2 Event Catalogue 64

4.2.3 Event Module Catalogue 65

4.2.4 Analysis and Checking 66

4.2.5 Code Generation 67

4.3 An Illustrative Example: File Access Control 67

4.4 Evaluation of the EventReactor Language 71

4.5 Similarities and Differences with Compose* 73

4.6 Future Work 74

4.7 Conclusion 76

5 Design of the Runtime Environment of EventReactor 77

5.1 Requirements for the Implementation of the Runtime Environment . 77

5.2 Data Structures 80

5.2.1 Representing Events at Runtime 80

5.2.2 Representing Event Modules and Publishers at Runtime . . . 81

5.3 Runtime Behavior 84

5.4 Illustration of Runtime Behavior 91

5.5 Runtime Behavior of Reactor Types . . . 101
5.6 Conclusion . . . 104

6 Multiplicity of Processes and Implementation Languages 105

6.1 Problem Statement . . . 106
6.2 Supporting Multiple-Process Java Software in EventReactor . . . 110
6.3 Supporting Distribution-Sensitive Specifications in EventReactor . . . 122
6.4 Conclusion and Future Work . . . 124

7 A Case Study for the Evaluation of the EventReactor Language 127

7.1 An Illustrative Runtime Enforcement Technique . . . 128
7.2 Implementing Recoverable Process in Imperative Languages . . . 131
7.3 Implementing Recoverable Process in Existing Runtime Enforcement Frameworks . . . 132
7.4 Implementing Recoverable Process in Aspect-Oriented Languages . . . 138
7.5 Implementing Recoverable Process in EventReactor . . . 145
7.6 Conclusion . . . 158

8 Conclusion and Future Work 159

8.1 Problem . . . 159
8.2 Solution . . . 160
8.3 Evaluation of the EventReactor Language . . . 163
8.4 Implementation Challenges . . . 165
8.5 Future Work . . . 166

Bibliography 169

List of Figures

2.1 The specification layer of the canonical model 14

2.2 The implementation layer of the canonical model 16

3.1 A meta-model for Event Composition Model 37

4.1 An overall view of the EventReactor compiler 63

5.1 The sequence of actions to detect and publish an event 85

5.2 The sequence of actions to identify event modules 87

5.3 The sequence of events to store a document 91

5.4 The runtime view of the document-editing software 96

6.1 An example causally-dependent sequence of events 107

6.2 An overall view of the EventReactor compiler with the support for distribution-transparent specifications 112

6.3 A runtime view for the distributed document-editing software 118

6.4 The runtime overhead for filling event module tables 120

6.5 The runtime overhead for processing events 121

7.1 The concepts of the Recoverable Process technique 129

7.2 An abstract block diagram of the media player software 131

List of Tables

3.1 Four possible instantiation strategies for event modules 39

3.2 Implementing Event Composition Model in the existing programming languages 53

5.1 The structure of the table *user_request.document_eventmodule* 95

Listings

4.1 Prolog facts to represent predefined events 57
4.2 The structure of event packages 59
4.3 The structure of reactor chains 61
4.4 The structure of behavioral specifications of reactor types 61
4.5 An example implementation of a reactor type 62
4.6 A specification of event module for the usage protocol of a file 68
4.7 A specification of the reactor chain for the usage protocol of a file . . 69
4.8 A specification of event module for the recovery actions 70
4.9 A specification of reactor chain for the recovery actions 71
5.1 A specification of event module for the document-editing software . . 93
5.2 A specification of reactor chain for storing a document 93
5.3 A specification of event module for the recovery actions 94
5.4 A specification of reactor chain for the recovery actions 94
6.1 Spanning the causal thread of execution to server-side 115
6.2 Spanning the causal thread of execution from client-side 115
6.3 A distribution-sensitive specification of event modules 123
7.1 A specification for the global recovery in JavaMOP 134
7.2 A specification for the local recovery of *UserInterface* in JavaMOP . . 136
7.3 A specification for the global and local recoveries 137
7.4 A specification of timing property for the global recovery of processes 138
7.5 An aspect representing the concept *AppProcess* 139
7.6 An aspect representing the process *MPCore* 139
7.7 An aspect representing the concept *RecoveryUnit* 140
7.8 An aspect representing the global recovery unit 141
7.9 An aspect representing the local recovery unit for *UserInterface* process 141
7.10 An aspect representing the concept *ProcessManager* 142
7.11 An aspect to implement *ProcessManager* for the global recovery . . . 144
7.12 An aspect to coordinate the global and the local recoveries 144

7.13 Declaring an event . . . 146
7.14 An excerpt of the class *MPCore* . . . 147
7.15 Declaring a publisher . . . 148
7.16 A reactor chain implementing the concept *AppProcesss* . . . 149
7.17 An event module representing the child processes of interest . . . 151
7.18 A reactor chain implementing the concept *RecoveryUnit* . . . 152
7.19 An event module representing the global recovery unit . . . 153
7.20 An event module representing a local recovery unit . . . 154
7.21 A reactor chain implementing the concept *ProcessManager* . . . 154
7.22 Event modules representing process managers . . . 155
7.23 Representing recovery constraints . . . 156
7.24 A reactor chain implementing an application-specific composition constraint . . . 156
7.25 An event module representing an application-specific composition constraint . . . 157

Chapter 1

Introduction

In today's practices, software is usually composed of multiple subsystems that are possibly developed in various languages and are distributed across multiple processes. The behavior of such software is affected by an increasing number of external factors since they are generally integrated in networked environments, interacting with many systems and users. As the complexity of software and its execution environment increases, ensuring that the software is failure-free becomes a challenge.

Several validation and verification techniques exist for this matter. However, due to the complexity of the software and its execution environment, it is economically and technically unfeasible to ensure that the software is failure-free. This motivates techniques that enable the software to tolerate failures and to continue operating in case of failures. An example is runtime enforcement techniques [9, 24] that check the actual execution of software against the formally specified properties of the software. If a failure is detected, diagnosis and recovery actions may be performed to respectively detect the causes of the failure and to recover the software from the failure.

There is a considerable number of frameworks [55, 27, 85, 17, 82, 66, 47, 64, 11, 12, 86, 60, 38, 71, 94, 10, 81, 26, 8] that can be employed to implement runtime enforcement techniques. However, they fall short in representing runtime enforcement concepts *naturally*. This thesis introduces Event Composition Model, which is a computation model for runtime enforcement frameworks to facilitate a natural representation of runtime enforcement concepts.

In the following sections, we first give motivations for using runtime enforcement techniques. Second, we discuss the contributions of the thesis, which are: a) the identification of the problems in natural representation of concepts, b) Event Com-

position Model, c) the EventReactor language as a realization of Event Composition Model, d) support for multiple-language software in EventReactor, and e) support for multiple-process software in EventReactor. The chapter concludes with giving information about the remaining chapters of the thesis.

1.1 Motivations

The work presented in this thesis was partially carried out within the TRADER [92] project, which aimed at developing methods and tools for ensuring the reliability of digital television (DTV) sets. In consumer electronic devices, the implementation of functionality is shifting from hardware to embedded software. Such software is formed around multiple layers. For example software modules that directly interact with hardware, operating systems, and drivers reside at the lower layers. At higher levels controllers and application software reside. Software modules may be developed in various languages and may be distributed across multiple processes, since different kinds of functions are involved.

To ensure the correct functioning of software, one may try to prevent faults in software during the development process. There are various different techniques that can be employed for this purpose in practice. Testing [39, 51], model checking [49] and static analysis [72] are examples. These techniques check the software correctness before it is delivered as a product.

When the television sets were largely implemented as analogue devices in hardware, the reliability of these sets mainly depended on assuring that the hardware was working properly. At that time, the features of the television sets were largely limited to a few functions. Nowadays, however, the shift from analogue hardware-based television sets to software-based ones with a lot of new and complex features, has made assuring reliability during the production process a huge challenge.

As a complementary approach to fault prevention and/or removal before the delivery, one may adopt fault-tolerance techniques as well. These techniques assume that not all faults can be detected and removed from software before it is delivered [6]. Therefore, in case of failures, *runtime enforcement techniques* [9, 24] are developed to enable software to continue with its operation while preserving its desired quality features.

The thesis considers runtime enforcement techniques suitable to detect and recover from the failures that occur in DTV's during their operational phase in their target environment.

There is a considerable number of frameworks [55, 27, 85, 17, 82, 66, 47, 64, 11, 12, 86, 60, 38, 71, 94, 10, 81, 26, 8] that can be employed to implement runtime enforcement techniques. In this thesis, software that is augmented with runtime enforcement characteristics is termed as *base software*.

In runtime enforcement, the active execution trace of base software is verified against its specified properties. If any deviation between these two is detected, it is considered a failure in the base software. In this case, the causes of the failure are diagnosed dynamically, and appropriate recovery actions are carried out to enforce the desired properties. Repairing the failures at runtime can be regarded as a distinguishing characteristic of runtime enforcement techniques compared to other verification and validation techniques.

To ease the design process, current runtime enforcement frameworks generally adopt specification languages that are used to define the properties of the base software, diagnosis rules and recovery actions. These specifications are also used to generate code rather than implementing the runtime enforcement functionalities manually.

Unfortunately, the specification languages employ the elements of the programming languages of the generated code, and therefore, they fall short in representing the runtime enforcement concepts *naturally*. By the term concept we mean a fundamental abstraction or definition that exist in most runtime enforcement techniques. For example, in object-oriented languages, objects and methods are represented as *first-class* abstractions. Therefore, if a specification language employs an object-oriented language as its underlying language, it can represent the notion of objects and method naturally. However, the other concepts such as processes, subsystems, and groups of these, cannot directly be represented in an object-oriented language. As a result, programmers must map these concepts to the existing elements of the languages. Such mappings may cause scattering and tangling of the runtime enforcement concepts in implementations. This reduces the modularity and compose-ability of the specifications of the concepts. Moreover, adoption of the elements of the underlying languages in specifications causes the specifications to become too specific to the employed programming languages and platforms. This reduces the reusability and comprehensibility of the specifications, and increase their complexity.

Providing *too abstract* specifications for runtime enforcement concepts may not be the desired solution, however. This will make the compilation of the specifications to an efficient code too difficult or even impossible. Moreover, the compiler has to make some assumptions how the abstract specifications must be mapped to the code; these assumptions may not be the choice of the software engineers.

It is also not feasible to suggest a complete new set of programming languages that are designed specifically for implementing runtime enforcement techniques. There

is a huge amount of legacy code implemented in various languages that cannot be disregarded.

To overcome the above mentioned problems, this thesis claims that there is a need of a new *computation model* that can be used to abstract the base software with a set of features that are natural in implementing the runtime enforcement techniques.

1.2 Contributions

To facilitate the natural representation of runtime enforcement concepts, this thesis defines a new computation model and its implementation that can be used to create an executable layer on top of the base software. The contributions of this thesis and the novel features of the computation model are briefly explained in the following subsections.

1.2.1 Identification of Problems in Implementing Runtime Enforcement Techniques

This thesis proposes a conceptual model termed as **canonical model**, which captures the common concepts among different runtime enforcement frameworks. The canonical model identifies the following core concepts in runtime enforcement frameworks: Specification of base software, specification of diagnosis, specification of recovery and specification of constraints.

To ease the code generation and integration process, the specification languages usually adopt the elements of their underlying languages. This, however, makes it difficult to have a natural representation of the concepts of interest. As such, the thesis observes the following shortcomings: a) decreased modularity, b) decreased abstraction level, and c) decreased compose-ability.

To represent the concepts of interest naturally, the thesis identifies the following requirements to be fulfilled by specification languages:

- **Modularization of specifications**: the specification languages must offer first-class abstractions that correspond one-to-one to the runtime enforcement concepts. In this case, the concepts can directly be represented by these languages, so the scattering and tangling problem is avoided.
- **Abstraction of specifications**: the abstractions offered by the specification languages must be defined naturally at the right level without incorporating

the implementation context i.e. implementation language and process structure, so that the portability and reuse of specifications are supported.

- **Compose-ability of specifications**: the composition language offered by the specification languages must offer a rich set of constructs to implement variable composition strategies. The composition must be able to a) integrate various specifications; b) deal with various elements in specifications; c) cope with different implementation languages and platforms of the base software; and d) facilitate constructing higher-level specifications by systematically reusing the existing ones.

To validate the canonical model and the identified requirements, the thesis elaborates on the existing runtime enforcement frameworks. Two categories of these frameworks are evaluated: a) the systems with dedicated specification languages and compiler; b) the languages supporting *embedded contracts*.

1.2.2 Enhancing Naturalness of Runtime Enforcement Concepts

The thesis claims that specification languages must provide first-class abstractions that respect the characteristic features of the concepts of interest.

The thesis identifies these features as follows: a) transient nature of runtime enforcement concepts; b) open ended-ness of the kinds of elements in specifications; c) no strict hierarchy among specifications.

The transient nature implies that the runtime enforcement techniques are derived by the changes of the system states. For example, the verification process observes and checks the changes that occur in the execution of the base software. When the verification of a property fails, it may trigger a diagnosis process. Therefore, a specification language must provide elements that represent the changes in the states of interest in each concept.

It is not easy or even possible to foresee all kinds of elements that are desired to be represented in the specification languages of runtime enforcement frameworks of today or in the near future. This implies that specification languages and their implementations must also be open-ended with respect to their elements.

There is usually no strict hierarchy among the specifications. A specification may be decomposed into sub-specifications; or the runtime enforcement concepts may also be regarded as the base software, so it must be possible to specify their proper-

ties. Absence of a strict hierarchy among specifications implies that a specification language must facilitate arbitrary composition of concepts with each other.

To achieve naturalness in the representation of the runtime enforcement concepts, the thesis introduces a computation model termed as Event Composition Model. In this computation model, the changes in the states of interest are termed as **event**. A set of relevant events are grouped as a linguistic abstraction, which is termed as **event module**. In the specification languages of runtime enforcement frameworks, we consider event modules more natural than adopting the elements of the underlying languages. Event modules are defined by the help of an **event composition language** that selects the events of interest. Event modules are identified by their unique names, they have input and output interfaces, and a set of implementations that are termed as **reactors**. Reactors have type and may publish their internal events to be used by other event modules. A set of related reactors that must process events in sequence is defined as a **reactor chain**. Event modules may be composed with each other; the composition constraints among event modules are defined by the **event constraint language**.

Event Composition Model respects the characteristic features of the runtime enforcement concepts in the following ways. Events represent the transient nature of the concepts. The model is open-ended for introducing new kinds of events and reactor types. This allows introducing new kinds of elements in specifications, when desired. The event composition language facilitates selecting events published by the implementations of event modules, and grouping them to define new event modules. This helps to form arbitrary hierarchies of event modules.

An important issue is how to implement Event Composition Model effectively, by using the current programming languages. For this purpose, the thesis evaluates the relevant programming languages with respect to their support for implementing Event Composition Model. The evaluation reveals that none of the existing languages can implement the model such that the desired quality attributes, such as modularity, abstractness and compose-ability are accomplished. Nevertheless, aspect-oriented languages offer promising features for this matter.

1.2.3 The EventReactor Language: an Implementation of Event Composition Model

The thesis proposes the EventReactor language that implements Event Composition Model. EventReactor provides dedicated linguistic elements to define event modules, reactors, reactor types and events. It makes use of the Prolog language as its event composition language.

EventReactor supports four sorts of predefined events in programs. These are the events that correspond to the following state changes: a) before invocation of methods; b) after invocation of methods; c) after invocation and immediately before execution of methods; and d) after execution of methods, which have terminated normally

EventReactor is extendable to support new sorts of events which are termed as user-defined events. It also provides an API to programmers for defining new sorts of reactor types. Dedicated operators are provided to compose event modules with each other and to specify the constraints among them.

The specifications are defined independently from any programming language, and the compiler of EventReactor facilitates generating code for Java, C and .Net languages. Reactors and reactor chains are parameterizable, and are defined separately from event modules. This increases the reusability of event modules and their implementations.

There are two basic ways in utilizing the EventReactor language: a) as an underlying language for the specification languages of runtime enforcement frameworks; b) as an implementation language of runtime enforcement techniques.

In the design of the EventReactor language, we are inspired by the aspect-oriented language Compose*. Since the Compose* language does not fulfil all the requirements in implementing Event Composition Model, it has to be extended considerably for this matter.

1.2.4 Supporting Multiple-Language Software

The systems of today are more and more implemented using multiple languages and this trend seems to continue also in the near future. For embedded systems, for example, it is quite common that the core of the system is implemented in C, applications in C++ and the user interface in Java. In addition, the common use of domain specific languages accelerates this development as well.

It is reasonable to assume that the runtime enforcement frameworks that can simultaneously work with multiple base languages will be preferable with respect to the ones that are designed for single language systems only.

The specification languages of runtime enforcement frameworks and the compilers must cope with the software implemented using multiple languages. The current runtime enforcement frameworks, unfortunately, fall short in supporting base software implemented in different programming languages.

In Event Composition Model, the event composition language facilitates selecting events from systems implemented in various languages. Since the EventReactor language does not make any assumptions about the implementation languages of the base software, event modules can be potentially reused for software developed in different languages. The multiple language support is one of the key characteristics of the EventReactor compiler.

1.2.5 Supporting Multiple-Process Software

It is now more and more usual that applications are designed to run on distributed system architectures. The common structuring concept in system design is based on the notion of a process. Naturally, it is preferable that runtime enforcement frameworks are capable of working on systems with different process structures.

Unfortunately, most runtime enforcement frameworks cannot be utilized for distributed systems. There are a few runtime enforcement frameworks that can work with distributed architectures. These systems, however, adopt specifications that contain information about the process structure. This increases the complexity and reduces the reusability of the specifications.

The EventReactor language and compiler facilitates *distribution transparency*. For distributed Java applications that make use of Java-RMI as middleware, the specification of event modules in EventReactor is transparent from the underlying process structures. For this purpose, the EventReactor compiler performs code analysis of the application software and facilitates selecting events from distributed software.

1.3 Thesis Overview

The thesis is organized as follows:

Chapter 2 defines a canonical model for the runtime enforcement frameworks, identifies the problems in the existing runtime enforcement frameworks and outlines requirements to overcome these problems. The chapter validates the canonical model and the requirements by elaborating on the existing runtime enforcement frameworks.

Chapter 3 defines the characteristic features of the concepts in runtime enforcement techniques. Accordingly, it introduces the computation model Event Composition Model, and discusses the suitability of the computation model to respect the char-

acteristic features. The chapter evaluates the existing programming languages with respect to their suitability to implement Event Composition Model.

Chapter 4 introduces EventReactor as the realization of Event Composition Model. The chapter explains the language and the compiler, and by means of an example illustrates the capability of EventReactor to define event modules.

Chapter 5 explains the runtime environment of the EventReactor language. For this matter, it explains the data structures and algorithms employed to represent events, event modules and to process the events.

Chapter 6 discusses how the EventReactor language and its compiler can cope with multiplicity of processes and languages. The solutions that are offered by the EventReactor language, such as distribution-transparent and distribution-sensitive specifications, are discussed.

Chapter 7 evaluates the suitability of the EventReactor language in providing natural representation of the runtime enforcement concepts. For this matter, it provides an implementation of a runtime enforcement technique called Recoverable Process in EventReactor. Recoverable Process, which is introduced in the TRADER project, is a technique for making processes fault tolerant. Chapter 7 discusses three implementations of Recoverable Process, which are in an imperative language, in an existing runtime enforcement framework and in an aspect-oriented language. The chapter illustrates the shortcomings of the existing frameworks and languages in representing the concepts of Recoverable Process naturally. Finally, the chapter discusses an implementation of Recoverable Process in the EventReactor language, and explains the suitability of the EventReactor language to overcome the identified shortcomings.

Chapter 8 provides conclusions and outlines the future work.

Chapter 2

Requirements for Runtime Enforcement Frameworks

Due to the complexity of todays software, several different validation and verification techniques are introduced since none of the available techniques can cover all facets of software to ensure its failure-freeness. **Testing** [39, 51], **model checking** [49], and **static analysis** [72] are examples.

In testing, software is executed to infer that, for example, it meets the requirements and/or does not have bugs. Testing can be performed manually by programmers or can be (partially) automated. Testing cannot assure that software is completely failure-free. Nevertheless, it can show that software fulfills a set of requirements under certain conditions.

In model checking, a model of the software is checked against the formally-specified requirements that the model or the software must fulfill. Counter examples may be reported by model checker if properties are not satisfied. The main challenges in model checking are defining expressive models, and dealing with the state space explosion problem in which number of states grows exponentially in the number of model variables. In general, the more expressive models are, the more likely is that the space explosion problem is experienced. Static analysis techniques check the program code without executing it; for example, to find coding errors and to check it against formally-specified properties.

The above-mentioned techniques check the behavior of software before the software is deployed in its target execution environment. However, the behavior of software is likely to be affected by the execution environment, for example due to the influence of networked programs, interactions with other systems, reaction of users, etc.

As complementary to the above mentioned techniques, **runtime enforcement** [9] is introduced to check the correctness of software in its actual execution environment. Runtime enforcement can be regarded as a fault-tolerance technique to enable software to continue operation in case of failure. We think that the term runtime enforcement has a more emphasis on the correction of errors at runtime whereas fault-tolerance, in some special applications, may aim at partial correctness or approximate results.

Runtime monitoring, runtime assertion checking, and runtime verification are also related to runtime enforcement in that they all enable verifying the behavior of executing software against its specification. In all these approaches, diagnosis and recovery actions can be added in case a failure is detected. In run runtime enforcement and fault-tolerance, diagnosis and recovery processes are considered as the necessary actions, whereas in other techniques, these may be seen as optional features. Nevertheless, in most cases, these terms can be used interchangeably. Within this context, it looks like that the use of a term is more like a personal preference than a fundamental choice. For the rest of the thesis, we will use the term runtime enforcement.

Runtime enforcement consists of three complementary processes: **verification**, **diagnosis** and **recovery**.

In this thesis, software that is augmented with runtime enforcement characteristics is termed as *base software*. In runtime enforcement, the active execution trace of base software is verified against its specified properties. If any deviation between these two is detected, it is considered a failure in the base software. In this case, the causes of the failure are diagnosed dynamically, and appropriate recovery actions are carried out to enforce the desired properties. Repairing the failures at runtime can be regarded as a distinguishing characteristic of runtime enforcement techniques compared to other verification and validation techniques.

An advantage of runtime enforcement is that the verification process is carried out in the actual execution context of software; this creates a more realistic context for verification. Runtime enforcement may also dynamically detect the causes of failures and recover the base software from the failures. In addition, runtime enforcement may scale-up better than other verification techniques such as model checking, because only the active execution trace is considered. On the other hand, only the failures that occur in the active execution trace can be detected. This may limit the usefulness of runtime enforcement techniques especially if all the potential failures must be detected. The other disadvantage of runtime enforcement is the overhead that it imposes on base software. In hard real-time systems, the overhead introduced by runtime enforcement can cause the base software to exceed its acceptable timing requirements.

To identify the basic concepts and the relationships among these, in the Section 2.1, the thesis proposes a *canonical model* of runtime enforcement frameworks. The model is termed as canonical because it captures the common concepts among different runtime enforcement frameworks.

In Section 2.2, the chapter outlines the problems of the existing runtime enforcement frameworks regarding their support for the *modularization*, *abstraction* and *composition* of specifications.

In Section 2.3, the requirements to overcome the identified problems are discussed. Finally, to show the expressivity of the canonical model and to illustrate the identified problems, Section 2.4 gives illustrative examples of runtime enforcement frameworks.

2.1 A Canonical Model for Runtime Enforcement Frameworks

The existing runtime enforcement frameworks usually adopt a two-layered architecture that is composed of a specification layer and an implementation layer. In the following, the canonical model is depicted from these two perspectives.

2.1.1 The Specification Layer

Figure 2.1 represents the concepts in the specification layer. By the term concept we mean a fundamental abstraction or definition that exist in most runtime enforcement techniques.

Specification of Base Software defines the expected and/or unexpected properties of the base software. The specification refers to the concerns of the base software, such as invocations to the methods, data values. The properties are usually specified in a logical formalism such as regular expression [33] and temporal logics [49]. The verification of the specified properties results in new information, for example, indicating whether the properties are satisfied or violated. This information can be used by the *Specification of Diagnosis* and *Specification of Recovery*.

Specification of Diagnosis defines the rules to diagnose causes of failures; for this matter, it may refer to the concerns in the base software and/or the results of verification process. The diagnosis also results in new information indicating the results of the diagnosis. *Specification of Recovery* defines a set of rules that describe

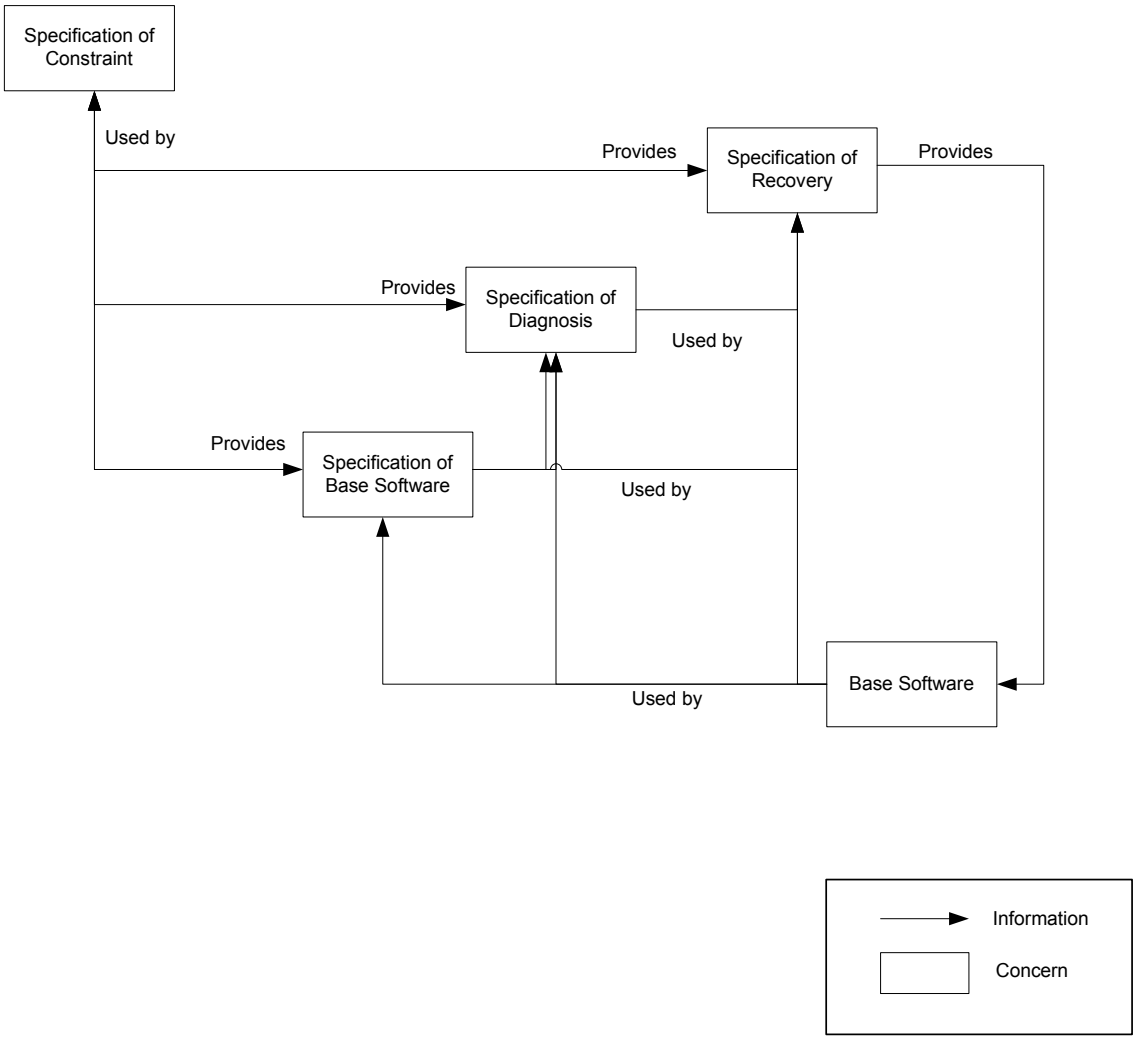

Figure 2.1: The specification layer of the canonical model

the recovery strategies, and for this matter it may refer to the specified information in other specifications.

Different recovery strategies may be employed based on the diagnosis results, fault assumptions, system characteristics, etc. Examples are [88]: compensation, backward recovery, and forward recovery. Compensation means that the system continues to operate without any loss of function or data in case of a failure. This requires replication of system functionality and data. Backward recovery (i.e. rollback) puts the system in a previous state, which was known to be failure free. Forward recovery (i.e. roll-forward) puts the system in a new state to recover from a failure. The recovery may also result in new data that are provided to the base software to heal it from the diagnosed failures. [1]

Specification of Constraints defines the interdependencies within and/or among the specifications of runtime enforcement techniques. For example, if there are multiple properties to be verified, it may be required to specify the order in which the properties must be verified. As another example, one may be interested in specifing that if a recovery action is being executed, the verification of other properties must be suspended until the base software is in a stable state.

Our definition of base software is general. As such, any software may be regarded as a base software including a runtime enforcement technique. This allows the definition of hierarchically organized runtime enforcement techniques.

2.1.2 The Implementation Layer

Since the specifications are more abstract than actual code, runtime enforcement frameworks provide a compiler that translates the specification to actual code and integrates the code with the base software. Figure 2.2 makes use of a UML component diagram [68] to represent the implementation layer. Here, the components are used to represent the concepts in the implementation layer rather than the implementation components.

There are five types of concepts: *Base Software*, *Verification*, *Diagnosis*, *Recovery* and *Constraint Controller*. The last four concepts are termed as runtime enforcement implementation concepts and can be regarded as the realization of the respective specifications shown in Figure 2.1. To ease the implementation effort and to

[1] Fault-tolerant computing and recovery techniques are active research areas. Our purpose here is not to give a comprehensive overview over these techniques, since it is out of the scope of this thesis.

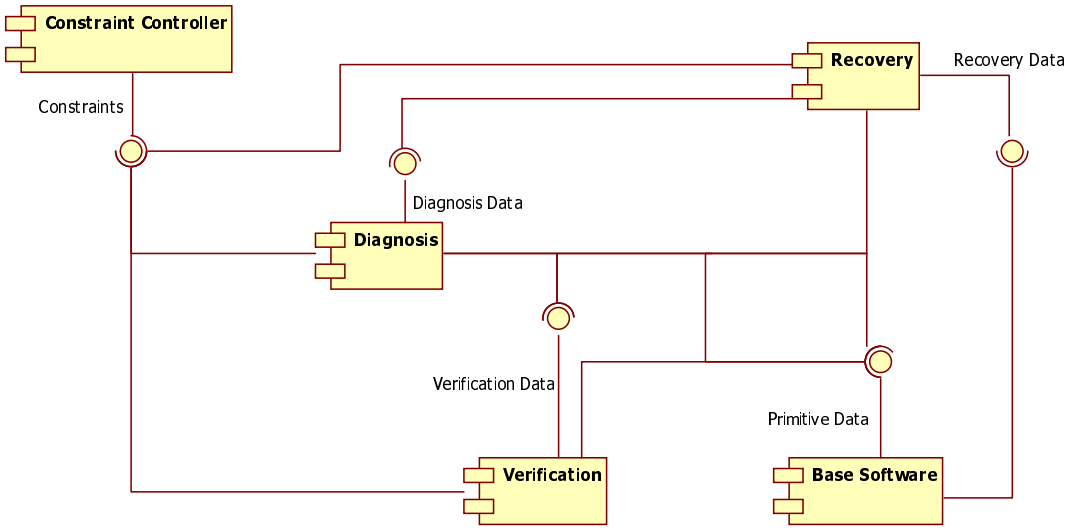

Figure 2.2: The implementation layer of the canonical model

avoid introduction of faults in realization, preferably runtime enforcement concepts must be generated from the specifications shown in the specification layer.

The concepts communicate with each other in various ways, for example, by publishing and receiving events, by invoking the services provided by the other concepts and by utilizing a shared memory to keep the data that must be exchanged among them. For example, *Constraint Controller* communicates with the concepts of interest to inform them about the controlling commands. In Figure 2.2, the communications are shown via the provided and required interfaces of the components.

2.2 Problem Statement

As it is discussed earlier, the existing runtime enforcement frameworks usually adopt a compiler that translates the specifications to code. This code can be expressed in the same language as the base software or it can be in an intermediate language, which abstracts the base software. In both cases, the generated code must be well integrated with base software from the perspective of runtime enforcement. We use the term **underlying language** to denote the language of the generated code.

The specification languages of the existing runtime enforcement frameworks usually adopt elements that are defined by their underlying programming language(s). For example, the systems [17, 38] that employ *aspect-oriented languages* [53, 4],

adopt the pointcut designators of these languages to specify the primitives of the base software; or in most of the existing runtime enforcement frameworks, the specifications of diagnosis and recovery are implemented in the underlying language(s) [55, 17, 38, 10, 26, 8, 81]. The specification languages may provide dedicated elements that are, for example, necessary to express properties in a formalism such as regular expression and temporal logic.

We observe two reasons why the specification languages of the runtime enforcement frameworks adopt the elements of their underlying languages. First, it would be otherwise too difficult or even impossible to generate code from the specifications if the specifications are defined at a very abstract level. Moreover, due to semantic gap between abstract specifications and underlying languages, the compiler should have to make certain assumptions how higher level concepts relate to low level elements; these assumptions might not always match the intentions of the software engineers. Second, a runtime enforcement technique consists of various concepts that have to be integrated with the base software. Adopting the elements of the base language in the definition of the runtime enforcement concepts makes the integration easier.

Despite its practical advantages, it may not be convenient to represent runtime enforcement concepts *naturally*, if the elements of the underlying language are adopted by the specification languages. Consider for example the following concerns of interest of the base software which may be regarded as the primitives of the specification: Objects, functions, calls, call patterns, processes, subsystems, layers, and groups of these. In diagnosis specifications, for instance, types of failures and data structures may be preferred to be represented explicitly. In case of recovery, various recovery strategies may be required to be specified. The underlying languages provide dedicated elements to represent some of the above listed concepts such as objects, functions, function calls; whereas the other kinds of concepts cannot directly be represented. Consequently, software engineers must provide a workaround representation for them using the available elements. We identify the following problems in such representations:

- **Decreased modularity**: The representation of a concept can be scattered across the multiple underlying language elements due to not directly representing the concepts of interest. Moreover, the representation of a concept may also get tangled with the representation of other concepts in the underlying language. Scattering and tangling are well-known problems that are discussed in the aspect-oriented literature [54]. They decrease the modularity of the concepts, decrease the reusability of the concepts and increase the complexity of software.

A typical example is that in most of the existing runtime enforcement frameworks, the specifications of base software properties, diagnosis rules and recovery strategies are tangled with each other in one specification module, and it is not possible to reuse a specified property, diagnosis rule and/or recovery strategy in other specification modules.

- **Decreased abstraction level**: The abstraction level of the specifications that adopt underlying programming elements can be considered too low level; this may reduce comprehensibility of the specifications. Moreover, the representation of concepts can become too specific to the underlying language. Consequently, it may be difficult to reuse the represented concepts for the software developed in different programming languages and/or in different platforms.

  A typical example is that in most of the existing runtime enforcement frameworks, the specifications of diagnosis rules and recovery actions are defined in the Java or C languages. Therefore, it is not possible to reuse them for software implemented in other languages.

- **Decreased compose-ability**: As shown in the canonical model in Figure 2.1, in runtime enforcement techniques, various specifications have to be utilized together to define the necessary concepts such as verification, diagnosis and recovery. We term this as the composition of specifications.

  We classify the composition strategies as **fixed** or **variable**. In the fixed composition strategy, the system supports a predefined composition mechanism. In the variable case, the composition strategy is programmable using a composition language.

  The fixed strategy can be too restrictive in one or more of the following cases: a) a large variety of verification, diagnosis and recovery concepts are required to be supported; b) base software is implemented in different languages and/or platforms; c) the application semantics demand tailor-able composition semantics.

  Adopting a variable strategy in composition is not trivial in general. The composition language must be rich enough to support the desired compositions in a flexible and effective way.

  In case the specifications adopt the elements of the underlying languages, the composition strategies can be required to be defined and implemented in the underlying languages as well. Similar to the previously defined problems, if the composition language does not support the composition strategies naturally, the complexity of the composition specifications may increase, and the comprehensibility and reuse may decrease.

A typical example for this case is that most of the existing runtime enforcement frameworks support a fixed strategy to compose specified recovery strategies with the specified properties of the base software. Here, the composition can only be performed upon the violation or validation of the properties. Therefore, they fall short in supporting more complex composition strategies, for example, conditional composition of recovery strategies with the properties.

2.3 Requirements for Runtime Enforcement Frameworks

To overcome the problems discussed in the previous section, we claim that a runtime enforcement framework must fulfill the following three requirements:

- **Modularization of specifications**: the specification languages must offer first-class abstractions that correspond one-to-one to the runtime enforcement concepts. In this case, the concepts can directly be represented by these languages, so the scattering and tangling problem is avoided. For example, the specification language must provide abstractions to represent the verification, diagnosis and recovery concepts as individual modules.

- **Abstractions of specifications**: the abstractions offered by the specification languages must be defined naturally at the right level without incorporating the implementation context i.e. implementation language and process structure, so that the portability and reuse of specifications are supported. For example, the specification language must facilitate specifying the verification, diagnosis and recovery concepts independently from the implementation languages of the base software.

- **Compose-ability of specifications**: the composition language offered by the specification languages must offer a rich set of constructs to implement variable composition strategies. The composition must be able to a) integrate various specifications; b) deal with various elements in specifications; c) cope with different implementation languages and platforms of the base software; and d) facilitate constructing higher-level of specifications by systematically reusing the existing ones.

  For example, the specification language must enable the specified verification, diagnosis and recovery concepts be composed with each other such that the overall functionality of a runtime verification technique is accomplished.

2.4 An Overview of Runtime Enforcement Frameworks

To validate the canonical model and to illustrate the identified problems, this section presents two categories for runtime enforcement frameworks with representative examples. The categories are: a) dedicated runtime enforcement frameworks explained in Section 2.4.1; and b) languages supporting embedded contracts explained in Sectionsec:embeddedREF. These systems differ from each other in their specification languages, in the realization of programming languages and/or process structures of the base software.

2.4.1 Dedicated Runtime Enforcement Frameworks

The runtime enforcement frameworks in this category provide a dedicated language to define specifications separately from the base software. A compiler is also provided to generate code for runtime enforcement concepts and to integrate the base software with the generated code. In comparison with general-purpose languages such as Java and C, dedicated languages are usually more declarative and as a consequence they simplify programming by focusing on *what* computation does but not *how* the program accomplishes the computation.

MaCS (Monitoring, Checking and Steering)

MaCS [55, 27, 85] provides three specification languages called PEDL (Primitive-Event Definition Language), MEDL (Meta-Event Definition Language), and SADL (Steering Action Definition Language). PEDL defines the primitive data that must be collected from the base software; MEDL defines the properties of the base software; and SADL defines the recovery actions. MaCS does not provide a specification language to express diagnosis rules. PEDL and SADL are dependent on the implementation language of the base software, where MEDL is language-independent. A Java-based implementation of MaCS is available.

PEDL specifies the objects of interest in the base software. It provides dedicated elements to specify events and conditions of interest that must be collected from the base software. Such events and conditions are defined in terms of methods and variables of the specified objects. An event corresponds to the updates of specified variables, the state when control enters a method, the state when control returns from a method, and/or when a specified condition becomes true or false. Primitive conditions are specified as Boolean-valued expressions over the specified variables.

Composite conditions are built from the primitive conditions using boolean connectives.

MEDL specifications import specified events and conditions from the PEDL specifications, use them to define composite events and conditions, and to define safety properties and/or alarms. The properties are expressed in an extension of the linear-time temporal logic [49] or in the regular expression formalism [43].

SADL defines recovery actions (i.e. **steering actions** in MaCS terminology) for the objects. A steering specification consists of two main sections: the declaration of base software objects that are involved in steering, and the definition of steering actions where the declared objects are used.

MaCS supports a fixed composition strategy by offering a fixed set of operators to compose primitive events and conditions with each other. In addition steering actions can only be composed with properties and alarms, when a safety property is violated or an alarm occurs; such compositions are specified within MEDL specifications.

MaCS provides a compiler to generate runtime enforcement modules from the specifications and to integrate them with the base software. The module *Verification* is executed in a separate process from the base software that is automatically instrumented to send the specified events and states to this process. The base software is also instrumented to execute the specified recovery actions.

MaCS falls short in fulfilling the identified requirements, because: a) its PEDL and SADL only facilitate specifying Java objects as the concepts of interest; b) PEDL and SADL incorporate implementation context, because they are specific to Java software; c) it offers a fixed and non-programmable composition language, and it does not facilitate defining specifications at higher-level of abstraction by reusing the existing specifications. For example, it is not possible to define a composite property by reusing and composing the existing properties.

MOP (Monitoring-Oriented Programming)

MOP [17, 82, 66, 47] is a framework for the development of runtime enforcement techniques. MOP aims at supporting software systems that are developed in various languages. JavaMOP and BusMOP are two available implementations of MOP. The former supports centralized Java programs, and the latter supports monitoring system buses using FPGA-based monitors.

JavaMOP makes use of AspectJ [53] as its underlying language. JavaMOP adopts the pointcut designators of AspectJ to specify the events of interest in the base

software. An event corresponds to the following state changes: when a method is invoked, when a method starts executing, when the invocation of a method or its execution terminates.

JavaMOP provides dedicated elements to express properties in various formalisms, and is extendable to support new sorts of formalisms. Currently Extended Regular Expression, Past-Time LTL and Future-Time LTL are supported. Java code blocks can be attached to a specified property, which are executed when the property is matched or failed at runtime. Such code blocks can be used to specify diagnosis and recovery actions.

The JavaMOP compiler translates each specification module to an aspect in AspectJ and some utility classes, which are further integrated with the base software using AspectJ compiler.

An illustration of the MOP shortcomings in fulfilling the previously-mentioned requirements is provided in Section 7.3 by means of an illustrative example. In the following, we provide a generic discussion of the shortcomings.

The modularization requirement is not fulfilled by JavaMOP, because a) the runtime enforcement concepts (e.g. primitives, properties, diagnosis and recovery) are defined in one specification module; b) it only supports the concept of object and groups of correlated objects. The abstraction requirement is not fulfilled because the specifications incorporate AspectJ and Java elements, so its abstraction level is close to program code.

Since JavaMOP is extendable with new formalisms, it offers a variable composition strategy for the composition of events with each other to form the properties of the base software. However, it offers two fixed operators termed *@fail* and *@match* for the composition of diagnosis and recovery specifications. JavaMOP offers *raw specifications* that require no formalism and allow programmers to write code in the target language (e.g., Java) to express the specifications. One may claim that it is possible to support variable composition strategy; however, this sacrifices the abstraction of specifications because they are basically the code in target language. JavaMOP also does not facilitate forming higher level specifications by reusing the existing ones.

PQL (Program Query Language)

PQL [64] checks if a centralized Java program conforms to certain design rules that are expressed as sequences of events associated with a set of related objects. The PQL language provides elements to abstract the program execution as a trace

of events. Each event contains a unique event ID, an event type, and a list of attributes. Eight types of events are represented: *field loads* and *stores*, *array loads* and *stores*, *method calls* and *returns*, *object creations*, and *end of program*.

The properties and recovery actions are specified in terms of PQL queries, which are patterns to be matched on the execution trace and actions to be performed upon a match. PQL supports two dedicated constructs *replace* and *execute* to define recovery actions. The former prevents a method to be executed and the latter invokes new methods. PQL does not facilitate specifying the specification of diagnosis rules.

A query may contain subqueries, which allow users to specify recursive event sequences or recursive object relations. Subqueries are defined in a manner analogous to functions of programming languages. They can return multiple values, which are bound to the variables in the calling query. By recursively invoking subqueries, each with its own set of variables, queries can match against an unbounded number of objects.

The PQL compiler translates the specifications to Java code that is inserted in the base software; it is executed in the same process as the base software.

PQL falls short in fulfilling the modularization requirement, because it only supports the concept of objects and the events that occur on objects. The abstraction requirement is not fulfilled because specifications are defined at the code-level and incorporate the elements of Java language. By the notion of subqueries, it is possible to form higher-level of specifications; however, the compose-ability requirement is not addressed by PQL, because it provides a fixed set of operators to compose events with each other in the definition of the specification of the base software, and to compose recovery actions with the specification of the base software.

Polymer

Polymer [11, 12] aims at enforcing security policies on centralized Java applications. It provides a language to define security policies that have three main elements: a) a side-effect free query method that determines how to react to security-sensitive method calls. Polymer uses the term action to refer to such method calls, and it makes use of a pattern-matching mechanism to allow programmers to summarize a collection of method calls as an abstract action; b) security state that can be used to keep track of the states of the software during execution; c) methods to update the security state of policies.

A query method returns one of six suggestions indicating that: the action is *irrelevant* to the policy; the action is *OK*; the action should be reconsidered after some other code is *inserted*; the return value of the action should be *replaced* by a pre-computed value; a security *exception* should be thrown instead of executing the action; or, the application should be *halted*. These are referred to as suggestions because there is no guarantee that the policy's desired reaction will occur when it is composed with other policies. These suggestions may be considered as recovery actions.

To support flexible but modular security policy programming, all policies, suggestions, and software actions are treated as first-class objects. Consequently, it is possible to define higher-order security policies that query one or more subordinate policies for their suggestions and then combine these suggestions in a semantically meaningful way, returning the overall result to the system, or other policies higher in the hierarchy.

Polymer does not facilitate specifying the specification of diagnosis rules. It also falls short in fulfilling the modularization requirement defined in the previous section, because it only supports the concept of objects and allows primitives be collected from the objects. The abstraction requirement is not fulfilled because specifications are defined at the code-level and incorporate the elements of Java language. It is possible to form higher-level of specifications by reusing the existing specifications; nevertheless the compose-ability requirement is not fully addressed by Polymer, because it provides a fixed set of operators to compose software actions with each other and to compose strategies with actions.

DIANA

DIANA [86] provides a language and compiler for the runtime verification of distributed Java programs. It proposes the PT-DTL formalism, a variant of past time linear temporal logic, that is suitable for expressing temporal properties of distributed systems. The properties expressed in PT-DTL are relative to a particular process and are interpreted over a projection of the trace of global states that the process is aware of. A property relative to one process may refer to local states of the other processes through remote expressions and remote formulae.

In order to correctly evaluate remote expressions, DIANA introduces the notion of Knowledge Vector and provides an algorithm which keeps a process aware of the local states of other processes that can affect the validity of a monitored PT-DTL formula.

To the best of our knowledge, the primitives that can be abstracted from the base software are events occurring on objects. The specification language of DIANA does not provide elements to define diagnosis and recovery actions. Since events that occur on objects are the only primitives that are supported by DIANA, the modularization requirement is not fulfilled by DIANA. The specification of properties is dependent on the process structure of the base software, so the abstraction requirement is not fulfilled. It offers a fixed set of operators, i.e. PT-DTL operators, to compose events with each other and does not facilitate forming higher-levels of specifications by reusing the existing ones.

MOTEL

MOTEL [60] facilitates the runtime monitoring of distributed software built on top a distributed object-oriented processing environment such as CORBA [22]. MOTEL provides a user interface through which users can select lists of objects and operations to be observed. Monitoring code is automatically generated and is inserted in the base software.

The monitoring is performed at the level of middleware, at the abstraction level provided by ODL/IDL specifications. Seven observable events are supported as primitives: *out request* occurs when an object (or the environment) requests the invocation of an operation on an object (target object); *in request* occurs when the request arrives at the target object; *accept request* occurs when the target object accepts the request and starts executing the operation; *out reply* occurs when the operation request has been processed and the target object is ready to transfer the result back to the source object; *in reply* occurs when the source object (or the environment) receives the results of the operation; *creation* occurs when an object is created; and *deletion* occurs upon the deletion of an object.

Similar to the other evaluated systems, MOTEL falls short in fulfilling the outlined requirements. It can only collect events from objects operating in a distributed environments. Consequently, the modularization and abstraction requirements are not fulfilled. It does not provide operators to compose events with each other, and to form higher-levels of specifications. Therefore, the compose-ability requirement is not satisfied.

RMOR

RMOR [38] is a runtime enforcement framework for centralized C software. It makes use of AspectC [4] as the intermediate language, and provides a pointcut designator

language similar to the one of AspectC, to express the specifications. An event corresponds to function calls or to variable assignments. Events may be composed with each other using Boolean operators. It is possible to share the specified events among multiple specifications.

The properties are specified as textual state machines. It is possible to define code blocks as C call-back functions, which are executed if the properties are violated. Diagnosis and recovery actions may be defined via such code blocks. The RMOR compiler translates the specifications to C code, which is inserted in the base software. The generated code is executed in the same process as the base software.

RMOR also falls short in fulfilling the modularization requirement, because it only supports the concepts of function calls and variable assignment, and only facilitates collecting event from these. The abstraction requirement is not fulfilled because the specifications incorporate the elements of the C language. Finally, the composeability requirement is not fulfilled, because it provides a fixed set of composition operators, i.e. Boolean operators and state machine; and it is not possible to define higher-level of specifications by reusing the existing ones.

2.4.2 Languages Supporting Embedded Contracts

Design by Contract [69] is an approach that recommends defining verifiable specifications for software modules, in terms of pre-conditions, post-conditions and invariants. The pre- and post-conditions respectively specify the properties that must be guaranteed prior to the execution and after the execution of some section of code. Invariants are properties that must always hold. Contracts may be checked statically or dynamically at runtime.

There is a set of technologies, referred to as embedded contracts, that extends an existing programming language with constructs to define pre-conditions post-conditions, and invariants, and to insert them into programs. Pre- and post-conditions are predicates that must respectively be *TRUE* before and after a block of code is executed. If a predicate must be *TRUE* before and after a block of code is executed, it is considered invariant. Programmers must identify the places in the base software code where the primitives of interest can be collected from, and embed the contracts there.

A benefit of the embedded contract is that the compiler and the existing features of the language can be reused, so tool developers do not have to duplicate the full compiler infrastructure, and the existing well-tested infrastructure can be reused to manipulate and analyze the target code. Another benefit is that users do not

have to learn a completely new language, compiler and IDE [29]. A drawback of embedded contracts is that the specifications are scattered across and tangled with software; consequently the complexity of software increases and the reusability of both software and specifications decreases.

The embedded contract technologies fall short in fulfilling the identified requirements, because of the following reasons. These technologies provides elements to represent contracts for methods and/or classes, but they do not provide elements to represent other concepts of interest such as processes; so the modularization requirements is not fulfilled. The contracts are specific to one programming language and they incorporate the elements of the programming languages; so the abstraction requirement is not fulfilled. The compose-ability requirement is not fulfilled also, because, these languages provide fixed operators to compose contracts with recovery code, and do not facilitate defining higher-level contracts by reusing the existing ones.

JASS (Java with Assertions)

JASS [10] is implemented for sequential, concurrent, and reactive software written in Java. It allows contracts be defined in the form of method pre- and post-conditions, class invariants, loop invariants. JASS also allows properties be expressed using universal and existential quantifications that range over finite sets. Trace assertions, based on the process algebra CSP [41], describe the observable behavior of a class and are used to verify the correct invocations of methods as well as the order and timing of method invocations.

If the specified properties are violated, a trace exception is thrown that can trigger a user-defined rescue block written in Java. A pre-compiler translates specifications to programs written in JASS into pure Java code, and inserts them in the base software.

APP (Annotation PreProcessor)

APP [81] provides linguistic constructs to define assertions and violation actions for C programs developed in Unix-based development environments. It also provides a compiler to translate the assertions into C code and inserts them in the base software.

The properties can be specified for function interfaces and/or functions bodies. For the specification of function interfaces, APP provides assertions to check consistency between arguments, dependency of return value on arguments, effect on global state,

the context in which a function is called, frame specifications, subrange membership of data, enumeration membership of data, and non-null pointers. For the specification of function bodies, APP provides assertions to check condition of the else part of complex if statements, condition of the default case of a switch statement, consistency between related data, and intermediate summary of processing. APP allows code blocks in C be attached to the properties. Such blocks can be employed to define recovery code.

Temporal Rover

Temporal Rover [26] supports software developed in Java, C, C++, Verilog, or VHDL. It supports the formalisms Linear-Time Temporal Logic (LTL) and Metric Temporal Logic (MTL) to express properties of the base software as future time temporal formulae as well as lower and upper bounds, relative-time and real-time properties. The assertions can be defined for methods, and recovery code can be written in the language of the base software.

Spec#

Spec# [8] extends C# with constructs for non-null types, pre-conditions, post-conditions, and object invariants. It supports the specification and reasoning about object invariants also in the presence of callbacks and multi-threading.

Spec# supports both static and dynamic checking of the specifications. For the dynamic checking, the Spec# compiler generates inlined code from the specified pre-conditions and post-conditions of methods. The inlined code evaluates the conditions and, if violated, throws an appropriate contract exception. Spec# supports inheritance of specifications, where a method's contract is inherited by the method's overrides. The run-time checks evoked by the method contract are thus also inherited.

JML

The Java Modeling Language (JML) [16] is a behavioral interface specification language for Java. It facilitates specifying class invariants, method pre- and post-conditions, frame properties, data groups, ghost and model fields.

Frame properties specify which parts of the system state may change as the result of the method execution. Any location outside the frame property is guaranteed to

have the same value after the method has executed (called the post-state) as it did before the method executed (in the pre-state).

In JML, it is possible to group a set of variables and refer to the group in the specifications, instead of referring to the variables individually. Such a group is termed as data group. JML supports the notion of model field, which is a specification-only field that provides an abstraction of (part of) the concrete state of an object. While a model field provides an abstraction of the existing state, a ghost field can provide some additional state, which mayor may notbe related to the existing state. Unlike a model field, a ghost field can be assigned a value.

Two sets of tools are provided for checking that JML annotated Java modules meet their specifications: Runtime assertion checking (RAC) tools, and Static verification (SV) tools. JMLC [18] is the main runtime assertion checking tool for JML. ESC/Java2 [20] is one of the existing static verification tools for JML specifications.

JML contracts may contain invocations to the methods. Such methods must be *pure* meaning that they are not allowed to have side-effects. Purity is statically checked by JML tools.

JML tools enforces behavioral subtyping [67], i.e. instances of a given type T must meet the specifications of each of type T's supertypes.

2.5 Summary

This chapter provides a canonical model for runtime enforcement frameworks, which illustrates the core concepts in the runtime enforcement frameworks and the relations among these concepts. Afterwards, the chapter claims that because the specification languages of the existing runtime enforcement frameworks employ the elements of their underlying programming languages, they fall short in representing various concepts such as processes, layers, subsystems, etc. Consequently, programmers must map these concepts to the elements of the underlying programming languages. Such mappings cannot always be modularized and become specific to the employed programming languages and platforms. These reduce the reusability and comprehensibility of the specifications, and increase their complexity. Moreover, the composition of concepts must be mapped to the elements of the underlying programming languages, which requires extra effort.

To overcome the identified problems, this chapter outlines three requirements that must be fulfilled by runtime enforcement frameworks: a) providing a rich set of first-class abstractions to represent various concepts modularly; b) representing the

concepts at higher levels of abstraction than the code level, so they can be reused from different implementation languages and process structures; c) providing a rich set of composition operators to facilitate expressing various composition strategies.

The chapter evaluates a representative set of runtime enforcement frameworks to illustrate their conformance with the canonical model and to discuss the degree to which they fulfill the identified requirements.

Chapter 3

Enhancing Naturalness of Runtime Enforcement Concepts

The aim of this chapter is to seek solutions to overcome the identified shortcomings of the existing runtime enforcement frameworks. To this aim, first the characteristic features of runtime enforcement concepts are defined in Section 3.1. Second, in Section 3.2 a computation model named as **Event Composition Model** is introduced.

The hypothesis of this chapter is that the proposed Event Composition Model can be a good basis in implementing runtime enforcement frameworks. To justify this hypothesis, Event Composition Model is briefly evaluated with respect to the characteristic features of runtime enforcement concepts in Section 3.3. In Sections 3.4, 3.5 and 3.6, a relevant set of current programming languages are evaluated to determine their suitability in implementing Event Composition Model. The result of the evaluation is summarized in Section 3.7.

3.1 Towards Natural Representation of Runtime Enforcement Concepts

The shortcomings of current runtime enforcement frameworks are discussed in Chapter 2. In particular, adopting underlying implementation language constructs in the specifications leads to three problems: a) reduced modularization due to scattering and tangling of runtime enforcement concepts in implementation; b) reduced abstraction level due to adopting the elements of the underlying languages in the

specifications; c) reduced compose-ability due to fixed composition strategies and/or due to the adoption of underlying languages in composition.

The source of these problems is due to the inability to specify and implement runtime enforcement concepts *naturally*. We think that a natural representation implies the following two requirements: a) the runtime enforcement concepts must be represented as the *first-class abstractions*[1] of the specification languages. b) The specification languages and their implementations must respect the characteristic features of the concepts.

A closer look at the existing runtime enforcement techniques lets us observe the following characteristic features:

- **Transient nature of runtime enforcement concepts:** The interactions among the concepts of runtime enforcement techniques have by nature a transient characteristic (i.e. are event-driven). For example, the verification process in the implementation layer of runtime enforcement techniques observes the changes that occur in the execution of the base software. These changes may correspond to calls on methods, returning of calls, assigning values to variables, etc. In other words, *changes in the states* of the base software are events that drive the verification process. This fact can be also observed in other concepts. For example, when the verification of a property fails, it may trigger a diagnosis process. Similarly, a recovery process may be activated as a result of the diagnosis process.

  The transient nature of the runtime enforcement concepts implies that a specification language must provide elements that represent the changes in the states of interest in each concept.

- **Open ended-ness of the kinds of elements in specifications:** The specifications that are supported by most of runtime enforcement frameworks can be briefly classified under four categories: specification of the base software, specification of diagnosis, specification of recovery and specification of constraints. Based on the requirements of applications, the specification languages define a set of linguistic elements to represent the concerns of interest. For example, for some applications, it may be sufficient to specify the properties of the functions in the base software. In some other applications, it may be necessary to specify processes, subsystems, or specialized kinds of these concepts such as highly available processes and subsystems.

[1] A language provides a set of first-class abstractions that are directly supported by the mechanisms of that language. A first-class abstraction may be passed as an argument of a call, it may be returned as a result of a call, it may be stored or retrieved, etc. First class abstractions are important factors in evaluating languages.

It is not easy or even possible to foresee all kinds of elements that are desired to be represented in the specification languages of runtime enforcement frameworks of today or in the near future. This implies that specification languages and their implementations must also be open-ended with respect to their elements.

- **No strict hierarchy among specifications:** Although Figure 2.1 shows a fixed hierarchy of specifications, for a general case, there should be no strict hierarchy among the specifications. For example, a system with diagnosis and recovery processes may be considered as base software as well. This results in multi-levels of runtime enforcement concepts.

  A specification may be decomposed into sub-specifications as well. For example, if the base software is composed of several subsystems, its specification may be decomposed in a set of sub-specifications, each one expressing the properties of a subsystem. In this case, it may be also preferable to organize diagnosis and recovery specifications per subsystem.

  The absence of a strict hierarchy among specifications implies that a specification language must facilitate arbitrary composition of concepts with each other.

3.2 Event Composition Model

This section introduces a computation model, which can help in implementing runtime enforcement techniques more effectively. The model is named as **Event Composition Model**. In the following subsection, we explain the abstractions in the computation model followed by a meta-model of the computation model.

3.2.1 The Abstractions in Event Composition Model

We term a change in the state of interest as **event**. In the dictionary [1], an **event** is defined as *"something that happens or is regarded as happening; an occurrence, especially one of some importance"*.

Definition of a state change can be different. For example, it can refer to an invocation of a method on an object, calling a function, begin or end of a thread of execution, a success or failure of a verification process, triggering a diagnosis process, committing a recovery action, etc. Nevertheless, in Event Composition Model, all such state changes are considered as events. We also do not assume a fix set of

events in the system. New kinds of events can be defined by application programs, or they can be introduced as libraries.

Although the notion of event seems to be a fundamental concept for runtime enforcement techniques, it is a too low-level representation with respect to the concerns of interest in specifications. For example, it may be necessary to represent all the events that are related to an object, a function, a thread of execution, a process, or a subsystem as linguistic abstraction. It is therefore logical to consider a group of *related* events as a *module*, which we term as **event module**.

In the literature [19], a module is defined as a software unit with input and output interfaces. The former defines the services that the module requires from its context; the latter specifies the services that the module provides for its context. A module promotes information hiding by separating its interface from its implementation.

We think that like the modules in programming languages, an event module must be uniquely identifiable for example by its name, must provide input and output interfaces and must separately specify its implementation.

The input interface of an event module is defined by the events that it groups. One important difference between the input interface of modules in programming languages and the input interface of event modules is that in programming languages input interfaces are invoked explicitly, whereas in event modules, invocations are *implicit*. An implicit invocation means that when the declared event occurs, the corresponding implementation is invoked without explicitly writing a code for it.

Events have **publishers**. For the set of related events that form the input interface of an event module, we consider their publishers also *correlated*.

The implementations of event modules are termed as **reactors**. Each reactor has a **type**, which defines the semantics of the reactor. Reactor types can be defined imperatively, by means of a program, or declaratively, by means of a domain specific reactor definition language. Each reactor type may publish new events during its operation. These are termed as **reactor events**.

Two sorts of reactor types are distinguished: **read-write** and **read-only**. In contrast to the read-write reactors, the read-only reactors cannot modify the states of software, so they do not have functional side-effects on software. The read-only reactor types can represent the concepts that must only collect information from the base software, without changing the base software. The verification and diagnosis concepts are examples. Recovery concepts can be represented as read-write reactors, since they modify the base software to heal the failures.

Each reactor has an input interface and an output interface, which are defined by a set of events. The input interface specifies a set of events of interest, which refers to zero or more events that are defined at the input interface of the corresponding event module. The output interface is the set of the reactor events that are published by the corresponding reactor type.

Reactors may be grouped in a module termed as **reactor chain**. Such reactors are composed with each other sequentially. It means that they process the events in a sequence starting from the first specified reactor within the reactor chain until the last reactor.

Zero or more reactor chains can be attached to an event module. The output interface of an event module is a union of the reactor events that are published by the reactors attached to the event module.

The selection of the events of interest, and the grouping of the events in an event module is carried out by an **event composition language**. The language is capable of selecting any event that is declared in the system and is in the scope. The reactor events can be selected and can form the input interface of other event modules. This enables the designers to create more abstract event modules by systematically composing the existing ones.

Event modules can be composed with each other at shared or non-shared events. In the former case, an event is part of the input interface of multiple event modules. The compositions may be constrained; the constrains is defined by using an **event constraint language**.

3.2.2 The Meta-Model of Event Composition Model

Figure 3.1 shows the meta-model of Event Composition Model to represent its elements and the relationships among them. This model is shown using the UML language.

Starting from the top, as the name implies, class *Event Module* represents event modules. This class has three outgoing associations and two incoming associations.

The outgoing association *input interface* refers to zero or more input events that are grouped by the event module.

The outgoing association *output interface* refers to zero or more events that are published by the reactors of other event modules.

Class *Event* is the base class for representing various sorts of events. Class *Base Software Event* is an example subclass that represents the events that are published by various elements in the base software. Examples of elements are objects, processes, and groups of objects.

Class *Reactor Event* represents the events that are published by reactors.

Various attributes are defined for each kind of event. We classify these in three categories: *StaticContext*, *DynamicContext* and *ReturnContext*. The static context of an event represents the set of attributes whose values do not change, and are used to select the event from the system. The name of an event is an example.

The dynamic context of an event represents the set of attributes whose values change at runtime. The unique identifier of the thread of execution in which the event is published is an example.

The return context of an event represents a set of attributes whose values are provided by the reactors that process the event. These are used to return values to the publisher of the event.

Every *Event* has a *Publisher*. The events forming the input interface of an event module may be published by various publishers. We consider such publishers correlated, as such their events are also related to each other and are grouped in one event module. Publishers are distinguished by their unique identifier.

Every *Event* is published in a thread of execution, which is shown by class *Thread*. By the thread of execution we mean an operating system thread, or a causal thread of execution that spans across multiple processes if the publishers of the events are distributed across multiple processes. The detail of maintaining such causal threads of execution is explained in Chapter 6. The threads of execution are distinguished by their unique identifier to which is assigned by the operating system or by the algorithm explained in Chapter 6.

The outgoing association *has* of class *Event Module* specifies that one or more reactor chains may be bound to an event module. The chains are represented by class *Reactor Chain*.

Class *Reactor Chain* shows that each chain is identifiable by a name, and has zero or more *Attribute*s and one or more *Reactor*s. Reactors can exchange information with each other via the attributes of the reactor chain.

Each *Reactor* specifies zero or more of the events, which are in the input interface of an event module, as its input interface.

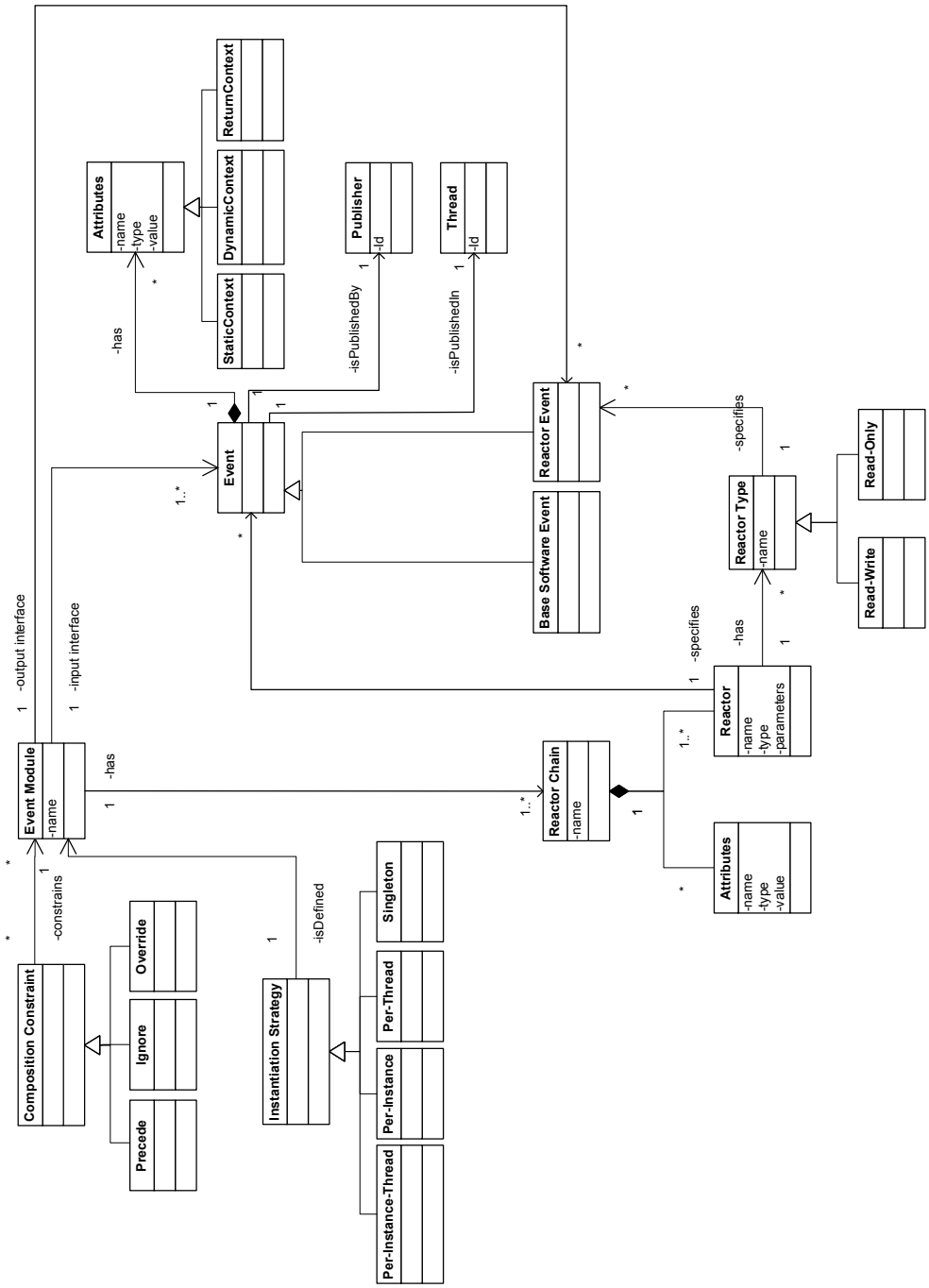

Figure 3.1: A meta-model for Event Composition Model

Each *Reactor* has a type, which is represented in the figure by *Reactor Type*. Two kinds of reactor types are distinguished: *Read-Only* and *Read-Write*.

Each *Reactor Type* publishes zero or more *Reactor Event* during its execution. These events form the output interfaces of event modules to which the corresponding reactors are bound.

The incoming association *constraints* of class *Event Module* refers to zero or more constraints among the event modules. We defined three constraints: *Precede*, *Ignore* and *Override*.

The constraint *Precede* specifies that if a selected event is a part of the input interface of two event modules A and B, the implementations of the event module A must react to the event first.

The constraint *Override* specifies that if a selected event is a part of the input interface of two event modules A and B, only the implementations of A must react to the event.

The constraint *Ignore* specifies that if a selected event is generated during the execution of reactors bound to the event module B, this event is ignored by the event module A.

The incoming association *instantiation* to class *Event Module* specifies the policy to instantiate event modules. Based on the publisher of the events, which form the input interface of event module, and thread of execution in which the events are published we consider four instantiation strategies. These are shown via classes *Per-Instance-Thread*, *Per-Instance*, *Per-Thread* and *Singleton* in Figure 3.1. An intuitive representation of these strategies is provided in Table 3.1, which is explained in the following.

The strategy *Per-Instance-Thread* means that we must distinguish both the publishers of the events, and the thread of execution in which the events are published. Here, individual instances of an event module must be created for each group of correlated publishers that announce events in the same thread of execution.

The strategy *Per-Instance* means that we must only distinguish the publishers of the events. Here, individual instances of an event module must be created for each group of correlated publishers, regardless of the thread of execution in which the events are announced.

The strategy *Per-Thread* means that we must only distinguish the thread of execution in which the events are published. Here, individual instances of an event module must be created for all correlated publishers, depending on the thread of execution in which the events are announced.

The strategy *Singleton* means that we must not distinguish the publishers nor the thread of execution in which the events are published. Here, a single instance of an event module must be created, shared among all the correlated publishers participating in the threads of execution.

Table 3.1: Four possible instantiation strategies for event modules

Publisher	Thread	Instantiation Strategy
1	1	Per-Instance-Thread
1	0	Per-Instance
0	1	Per-Thread
0	0	Singleton

3.3 Motivations for Adopting Event Composition Model

This section provides a brief evaluation of Event Composition Model in addressing the shortcomings of current runtime enforcement frameworks. A more detailed study will be presented in the following chapters. For this purpose, we investigate if event modules can offer a natural way in expressing the concepts of runtime enforcement techniques. To this aim, we refer to the characteristic features of the concepts that are described in the previous section:

- **Transient nature of runtime enforcement concepts:** We consider the notion of event fundamental in supporting the transient nature of runtime enforcement concepts. In Event Composition Model, input-output interface specifications of event modules and reactors are defined as events.
- **Open ended-ness of the kinds of elements in specifications:** In Event Composition Model, we do not assume a fix set of events in the system. This means that if needed, new event modules can be introduced to represent the emerging concerns of interest.
- **No strict hierarchy among specifications:** In Event Composition Model, since reactor types can publish events, and the event composition language can select these events as well, it is possible to compose event modules with each other such that a hierarchy of event modules can be formed.

In order to conveniently implement the presented computation model, preferably the implementation language must fulfill the the following requirements:

- Declaration of arbitrary state changes as an event; this provides flexibility for runtime enforcement specification languages in defining the concerns of interest.
- Implicit invocation on the event module implementations (reactors) upon the occurrence of events.
- An event composition language with the following properties:
  - Declarative so that explicit reasoning over specifications is eased.
  - Selection of any event that is declared in the system and in the scope, so that event modules of interest can be flexibly defined.
  - Explicit naming of the group of events that are selected, so that event modules can be defined and referred to by name.
- Definition of a set of reactor types with the following properties:
  - From a reactor type, one or more instances can be created and named. This enhances the reusability of types.
  - A reactor instance can be bound to an event module through the use of an explicit language keyword or statement, so that flexibility in defining the semantics of event modules are accomplished.
  - Encapsulates its own implementation, so that modularity is supported.
  - Reactor types may be programmed by their domain-specific specification languages, so that the semantics of reactors can be expressed in their natural form.
  - Variety of composition strategies for events can be expressed as domain-specific reactor types, so that strategies can be represented in their natural form rather than in general-purpose imperative languages.
  - A reactor type can publish its own events, so that new event modules can be formed using the existing ones.

3.4 Implementing Event Composition Model in Object-Oriented Languages

In the previous sections, we have informally introduced the concept of event modules as a possible way to overcome the shortcomings of the programming languages in implementing runtime enforcement techniques. We have also briefly motivated the

proposed approach by evaluating it with respect to the characteristic features of runtime enforcement concepts. We will now evaluate object-oriented languages to infer whether they fulfill the requirements mentioned in the previous section to implement Event Composition Model effectively.

Object-oriented programs are formed around a set of objects that specify real-world entities and the operations that can be performed on them [7]. An object is composed of a set of attributes and methods that group a set of statements. Methods inspect and update attributes. Classes abstract over objects that have similar attributes and methods. A class may implement one or more interfaces. Classes may be related to one another by dependency, inclusion (inheritance) or by containment (aggregation). In the dependency relation, the methods of a class invokes the methods of another class; invoking a method is also termed as sending a message. In the inclusion relation, one class is a subclass of another one. In the containment relation, objects of one class contain objects of another or the same class.

For a straightforward implementation of Event Composition Model in object-oriented languages, we assume that there is an interface which defines a list of methods, and there are classes which implement that interface. We assume that the methods defined in the interface form the input interface of an event module. The classes represent reactors. Since classes are data types in object-oriented languages, the reactors are also typed.

To bind multiple reactors to an event modules, multiple classes that implement the interface must be defined. Classes may publish new events by invoking the methods that are defined in an interface. Various sorts of events can be encapsulated and represented as methods (i.e. input interface of an event module).

For the following reasons we claim that object-oriented languages do not fulfill the requirements mentioned in the previous section; hence, cannot effectively implement Event Composition Model.

First of all, event module interfaces are activated by the occurrence of the corresponding events, whereas the methods of an object must be invoked explicitly.

Second, an event module is formed dynamically through the execution of an event composition program, while the interface of an object is fixed by its class definition.

Third, in the case of event modules, it is possible to bind an implementation to a group of events that form an event module. If this mechanism has to be implemented by an object-oriented language, an application must be written to select the corresponding events and linked these to their implementation code. This causes scattering and tangling in the object-oriented implementation. We assume that in

event modules, binding implementations to a group of events is supported by an explicit language keyword, and therefore scattering is avoided.

Fourth, the use of interfaces is not obligatory in all object-oriented languages. Therefore, an object-oriented implementation of an event module might not offer the separation of the interface and the implementations of the event module.

Fifth, although it might be possible to program various composition strategies, the implementations would be expressed in imperative object-oriented programs, rather than in a domain-specific composition language.

3.5 Implementing Event Composition Model in Aspect-Oriented Languages

The key concepts in aspect-oriented programming [54, 13, 75, 58] are join points, pointcut designators, advices and aspects. Join points are identifiable state changes in the execution of programs. Examples are execution of methods, creation of objects, and throwing of exceptions. Pointcut designators are linguistic constructs for querying join points from the program.

Advice is a program code that is executed when the corresponding join point is activated. An advice is bound to a set of join points through a pointcut designator. In most aspect-oriented languages, the combination of an advice and its pointcut designator is called an aspect.

The join point model of aspect-oriented languages defines the join points available for writing aspect-oriented programs, which are fixed in general.

Aspect-oriented languages introduce aspects as units of modularization for the code that scatters across and tangles with other program pieces in case object-oriented and/or procedural languages would have been used. Aspect-orientation promotes better separation of concerns than object-oriented programming, in case the scattered and tangled code can be abstracted as an aspect.

Let us assume that we implement event modules using aspects. In this case, the input interface of an event module may correspond to the join points selected by the pointcut designator of an aspect. A pointcut designator has a similar function as the event composition language, and the advice is similar to the implementation of event modules (reactor).

Aspect-oriented languages seems to provide a suitable basis for implementing Event Composition Model. To elaborate on this, we need a more detailed look at the cur-

rent widely known aspect-oriented languages. These are discussed in the following sections.

3.5.1 AspectJ

An illustration of the AspectJ shortcomings in supporting Event Composition Model is provided in Section 7.4 by means of an example. In the following, we provide a generic discussion of the shortcomings.

The join point model of AspectJ [53, 56] defines a rich set of events corresponding to method calls and executions, constructor calls and executions, field get and set operations, pre-initializations, initializations, static initializations, handler and advice executions.

The declarative pointcut designator language of AspectJ facilitates selecting events based on the static and dynamic information of join points. The static information can be for example the method names, and the dynamic information may refer to the arguments of the selected methods. Advices in AspectJ are pieces of Java code executed upon the activation of the corresponding join points.

Although event modules and aspects in AspectJ show similarities, there are also important differences. Firstly, events in Event Composition Model are assumed to be open-ended; new kinds of events can be introduced if necessary. The events in AspectJ are defined by its join point model and they are fixed.

Secondly, the pointcut designator of AspectJ is rather limited and mainly used for selecting the join points. If more complex operations are needed, such as grouping of join points, these have to be realized in the implementation of aspects (in advice code).

Thirdly, in event modules, arbitrary implementations may be bound to event modules using an explicit language keyword. In AspectJ different implementations can be bound to pointcut designators through subclassing of aspects or by writing an application code that use multiple aspects.

Fourthly, the implementations of event modules, which are called reactors, have types. We assume that reactor types can be specified using their domain specific (declarative) languages. In AspectJ, however, advices are expressed in an imperative Java code.

Fifthly, the pointcut designator language of AspectJ provides a fixed set of operators to compose events, to define variety of composition strategies, one must implement

the strategies as advices. In this case, composition strategies are implemented in an imperative language, rather than a domain-specific composition language.

Finally, reactors can publish arbitrary events. These define the output interfaces of the event modules, which can be selected and composed into new event modules if necessary. These events are treated in the same way as the events that are generated through the execution of programs. In AspectJ, however, advices do not have types and are regarded as anonymous methods. The pointcut designators of AspectJ are limited to select events that are published by all advices defined in an aspect without the possibility of distinguishing advices. An application code can be provided to partially address this deficiency. For example, one may try to rewrite an advice by putting its original body in a method and invoke this method from within the advice. By this way, an AspectJ pointcut specification can be used to select the event that is published by the advice. As it is studied by [79], this approach however, cannot deal with advices that utilize contextual information. Also, the complexity of software increases due to the additional application code.

3.5.2 AspectJ Extensions

In the literature, a number of extensions to AspectJ have been published to overcome some of its shortcomings.

To enhance the AspectJ pointcut designators with history information, Tracematches is introduced [76]. Tracematches adopts the regular expression formalism over AspectJ pointcuts to express history-based information. The history-based pointcuts of Tracematches are more expressive than the one of AspectJ, because it supports selecting events from multiple objects and allows representing a sequence of events occurring in a group of objects. In Event Composition Model, the event composition language is more general than the pointcut designator of Tracematches, because in Tracemacthes input interface of aspects are limited to a regular expression formalism.

In AspectJ, the binding of advices to objects is limited. It is either possible to bind a singleton instance of an aspect to all the selected objects, or a separate instance of an aspect is bound per object. Association Aspects [84] has been introduced to enhance AspectJ pointcut designators with a possibility to flexibly bind an advice to a designated group of objects. In Association Aspects, because an aspect is parameterized with a fixed-length group of objects, it is not possible to bind an aspect to arbitrary groups of objects. In Event Composition Model, the event composition language facilitates selecting arbitrary groups of events from one or more objects and allows attaching implementations to these.

The current AspectJ system is developed to work on a single-process Java platforms. To offer aspect-oriented programming for distributed Java implementations, DJCutter [73] is developed. DJCutter extends AspectJ with remote pointcut designators that can identify join points in the execution of a program running on a remote host. In DJCutter, advices are executed in a process called Aspect Server. In Event Composition Model, the event composition language must be able to select events generated by programs implemented in different programming languages, processes and possibly running at different hosts.

There have been various attempts to extend the fixed join point model of AspectJ. One may decorate, for example, the program code with annotations, and with the help of annotation based pointcut designator, these code can be selected. However, in AspectJ, the use of annotation is limited to certain language elements only.

The other way is to extend the join point model of AspectJ with new elements. For example in [37], the so called *loop join point* is introduced. Unfortunately, each such extensions do not change "the fixed set of elements feature" of the join point model.

In the language EJP [42], a new join point can be declared as a special kind of method interface. A declaration specifies a name for the explicit join point along with a return value, formal parameters, and a throws list. In the code where a join point must be activated, a reference to the corresponding join point declaration is made. In EJP, the implementation of a join point can be scattered to the different locations in the code. Moreover, the activation of a join point is explicit. In Event Composition Model, event publication is implicit.

IIIA [89] aims to extend the AspectJ join point model through the notion of typed join points. Such a type defines a set of context variables and a pointcut predicate. Individual classes in the program may bind to a declared join point explicitly with the use of a dedicated keyword. In IIIA, similar to EJP, the implementation of a join point can be scattered to different locations in the code. Also, the activation of a join point is explicit.

3.5.3 Compose*

Compose* [21] is a language- and platform-independent aspect-oriented language, which supports the languages C, Java and the .Net languages. The Compose* language is mainly a composition language and can be used to enhance various different implementation languages. The join point model of Compose* includes the events that correspond to the incoming and outgoing messages to and from objects. To react on the incoming and/or outgoing messages, Compose* defines the notion

of *filters* that are attached to objects. Each filter has a type that implements its functionality. The pointcut designator in Compose* is termed as a superimposition specification, which is expressed using a Prolog [90] program. A superimposition specification selects the objects of interest and binds the corresponding filters to these objects. Superimposition specifications are defined separately from filters; this increases the reusability of filters due to flexibility in bindings.

A straightforward way to implement event modules in Compose* is to represent an event module via filters and superimposition specifications. Here, filter types are means to define reactor types, and superimposition selectors are means to select the events of interest and bind an instance of a filter type to them. The pointcut designator of Compose* is expressed using the declarative Prolog language, and it facilitates selecting events from programs written in different languages. Dedicated different filter types can be employed to define various different composition strategies for events. Some filter types in Compose* are specified using their domain specific languages.

Despite its advantages, the Compose* language has some important shortcomings. First, the join point model of Compose* supports a fixed sorts of events, which are the events that correspond to incoming and outgoing messages. In Event Composition Model, however, events may refer to any state-change in the execution of the program.

Second, the event composition language of Compose* can only select the events that are activated per object. Therefore, it falls short in grouping events activated by arbitrary groups of objects, as it is defined by Event Composition Model. This also limits the possibility of selecting events from a group of objects that are distributed across multiple processes and/or are implemented in different languages.

Third, the filters in Compose* do not publish their events of interest. As such, it is not possible to compose event modules.

Finally, since events in Compose* are defined per object, distribution of events to different hosts is not supported.

3.5.4 AWED

AWED (Aspect With Explicit Distribution) [71] is an aspect-oriented extension to Java. The AWED language provides a dedicated pointcut designator to select events that may occur at different hosts. In addition, the pointcut designator can express history information; sequences of events are defined in terms of transitions of non-deterministic finite-state automata; The events can involve different hosts. It is

possible to specify a transition in the automaton that can trigger advises, which can execute remotely.

AWED partially fulfills the requirements in implementing event modules due to its support of distribution. However, AWED has some shortcomings as well. Firstly, it offers a fixed join point model that supports mainly method invocations. Secondly, similar to AspectJ, AWED does not separate implementation of aspects (advice code) from their interfaces (pointcut designator). Moreover, the pointcut designator cannot select the events that occur in advice code. By means of history-based pointcuts, AWED facilitates selecting events from multiple objects; however similar to Tracematches, the input interfaces of event modules can only be defined in terms of the history-based pointcut expressions that are available. Similar to AspectJ, AWED falls short in supporting varierty of composition strategies.

3.5.5 AspectWerkz and EOS

AspectWerkz [5] and EOS [78] are two languages that aim at unifying aspects with classes. The pointcut designators of these languages can select events that can be activated in both classes and aspects. This makes it possible to implement hierarchically organized aspects.

In AspectWerkz, aspects are defined as normal Java classes, and methods are used to define advices. However, a class representing an aspect must fulfill certain restrictions such as providing a set of methods with certain properties. AspectWerkz separates interfaces (pointcuts) of aspects from their implementation (advices). The other characteristics of AspectWerkz are similar to the ones of AspectJ, and therefore it has similar shortcomings in implementing event modules.

EOS, which is an extension to C#, also unifies aspects and classes such that an aspect can explicitly be instantiated. Advices are defined as normal methods without any constraint. This makes EOS more flexible than AspectWerkz. Similar to Association Aspects, EOS supports selecting events from a group of objects. The other characteristics of EOS is similar to the ones of AspectJ, and therefore it has similar shortcomings in implementing event modules.

3.6 Implementing Event Composition Model in Languages Supporting Implicit Invocation Mechanisms

In implicit invocation, instead of invoking a procedure directly, a procedure publishes one or more of its events. Other procedures that are interested in that event can register for the event. When an event is published, the implicit invocation mechanism invokes the procedures that are registered for that event. In this manner, the publisher of the event is decoupled from the procedures that react to the event.

There are a number of systems and languages that offer implicit invocation mechanisms. For example, Java virtual machine provides a tool interface that allows native libraries to select the events of the virtual machine and to control the Java virtual machine [50] accordingly. Similarly, the Linux operating system publishes events that can be processed by applications, in response to a condition such as memory segment violation and process destruction [59]. Also, some databases provide notations for defining active data triggers [32].

Apart from aspect-oriented languages, there are also languages or language extensions that support implicit invocations by providing dedicated elements to define events, to publish events and to react to events.

3.6.1 C#

C# provides the event-delegate mechanism [15] to define events and publish them. Events are defined as special class members and are published explicitly from within the class. Methods that are interested in an event must register for it. It is also possible to unregister from an event.

Let us assume that an event module is defined as a class in C#. Here, the class must implement the functionality to register for the events of interest that form the input interface of the event module, and to group them as it is desirable. The implementations of event module can be provided as methods in the same or in different classes. The implementations are explicitly invoked by the event module class.

There are the following shortcomings in implementing of event modules in C#. First, there is no (declarative) language to select and group events; consequently, the input interface of event modules cannot be expressed directly. Second, the invocation of implementation is explicit. Third, the implementations of event mod-

ules are expressed in imperative C# code, whereas in Event Composition Model the implementation of event modules can be specified using their domain specific (declarative) languages. Fourth, the separation of the interface and the implementations of event modules is not directly supported by the languages, and it depends on the way programmers implement the event modules. Fifth, a fixed composition strategy is supported for events.

3.6.2 EventJava

EventJava [28] is an extension to Java with a generic support for event-based distributed programming. EventJava introduces the notion of event methods that are a special kind of asynchronous method, and are used to define application-specific events. The invocation of event methods is a means to publish events. An event method can represent a single event or a group of events. An event method may have a set of formal arguments that are used to maintain the attributes of the corresponding event. EventJava considers a set of implicit attributes for each event, which convey contextual information.

Let us assume that an event module is implemented as an event method. In this case, a state change can be represented as a dedicated event method, and correlated events can either be grouped using an array-like constructs, or they can be composed with each other to form a sequence. The body of an event method defines the implementation of the event module. The implementation of an event module can publish new events.

There are the following shortcomings in implementing event modules in EventJava. First, events are published explicitly by program code. Second, the input interface of an event module is fixed by the definition of the corresponding event method in EventJava, because it is not possible to select events. Third, the event composition language provides a limited set of operators to group and compose events; whereas, in Event Composition Model an expressive domain-specific language is required to define variety of composition strategies. Fourth, the interface of an event module is not separated from its implementation, because, it is defined as one event method. Fifth, the implementation of an event module is imperative Java code, whereas, in Event Composition Model domain-specific reactor types are desirable.

3.6.3 EScala

EScala [35, 36] is an extension to the language Scala [74] with object-oriented events. EScala adopts the notions of join points and pointcut designators to facilitate the

selection of events. In EScala, events are declared as class members that can be accessed as attributes of objects. The definition of an event can refer to the state of the objects, to the relationships of objects with other objects, and to the events declared in the public interfaces of other objects. Events can either explicitly be published, or they can be selected from the software by the available pointcut designators. In the latter case, only method calls are available as events.

Let us assume that an event module is implemented as a class in EScala. In this case, the methods of the class must either explicitly register for the events that they are interested in, or the events of interest can be selected via pointcut designators. The methods in the EScala class provide an implementation for the event module. We identify the following problems in implementing event modules in EScala:

Firstly, events in Event Composition Model are assumed to be open ended; new kinds of events can be introduced if necessary. However, in ESclala the events that can be selected by the pointcut designators are fixed by the join point model of EScala.

Secondly, Event Composition Model expects expressive languages for the selection of arbitrary events, group them and define variety of composition strategies. However, the pointcut designator of EScala has a limited expressive power for this matter.

Thirdly, the separation of the interface and the implementations of event modules is not directly supported in EScala, and it depends on the way programmers implement the event modules.

Fourthly, the implementations of event modules, i.e. reactors, have types and can be specified using their domain specific (declarative) languages. In EScala, however, methods are expressed in the imperative Scala language.

3.6.4 Ptolemy

Ptolemy [77] is an expression language built on top of Java, which positions itself between implicit invocation mechanisms and aspect-orientation. It allows the execution of arbitrary expressions be identified as events, and facilitates defining event types to abstract over such events. An event type has a name, a return type and is composed of a set of context variable declarations. These context declarations specify the types and the names of reflective information exposed by conforming events. In Ptolemy, events must explicitly be announced, by binding an event type to an expression in the base program. Ptolemy allows handlers to be declaratively registered for a set of events using one succinct pointcut designator in a binding expression.

Let us assume that an event module is implemented as a class in Ptolemy. In this case, by means of pointcut designators, the class can select the events of interest and bind an implementation to them. We identify the following problems.

Firstly, because an event type must be explicitly bound to expressions in the programs, the announcement of events is explicit. Secondly, the pointcut designators of Ptolemy are not expressive enough to select arbitrary events and to compose or group them; it only facilitates the composition of events via disjunction operator. Thirdly, Ptolemy does not support domain-specific implementation of event modules. Fourthly, the input interface of an event module is not separated from its implementation, because pointcut designators and handlers are defined in one class.

3.7 Summary of the Evaluation

Table 3.2 summarizes the strengths and the shortcomings of the evaluated programming languages in supporting Event Composition Model. Here the abbreviation ECL stands for Event Composition Language. The first column represents the features that are expected to be supported by a language; these are taken from Section 3.3. The other columns in the first row list the evaluated languages. The characters *'+'*,*'o'* and *'-'* mean that the language fully supports, to some extent supports, and does not support the listed feature.

As it can be seen in the table, none of the evaluated languages fully provide the desired features of Event Composition Model. A detailed discussion of the shortcomings are provided in the previous sections. In the following, we provide a brief explanation of the cells that are marked as *'o'* for a selected set of languages that are considered relevant.

The cell "*Separation of interface and implementation of event module*" is marked as *'o'* for the object-oriented languages, because not in all the object-oriented languages the separation of interfaces from implementations are compulsory and/or possible. For example, in Java it is optional to define interface for a class.

The cell "*Publishing events of implementations*" is marked as *'o'* for AspectJ, because the pointcut designators of AspectJ provides a limited support to select events occurring from within advices. The cell "*Separation of interface and implementation of event module*" is marked as *'o'*, because even if pointcuts are defined separately from advices, the composition of advices with pointcuts is defined as a part of the advice.

The cell "*Implicit invocation*" is marked as *'o'* in AspectJ extensions, because in EJP and IIIA, events must explicitly be published from within the program.

The cell "*Selecting events (from multiple languages and processes)*" is marked as *'o'* for Compose*, because Compose* does not facilitate selecting events from multiple-process software. The events can be selected from a single object; therefore, although Compose* supports multiple base languages, it falls short in selecting events from multiple objects implemented in different languages. AWED also falls short in selecting events from languages other than Java.

The cell "*Variable event composition strategies*" is marked as *'o'* in Compose*, because Compose* facilitates defining new composition operators via dedicated filter types. However, the composition is limited to the event published by a single object.

The cell "*Explicit naming and grouping in ECL*" is marked as *'o'* in EOS, because EOS supports selecting events from a group of objects, however, it is not possible to bind an aspect to arbitrary groups of objects.

The cell "*Selecting events (from multiple languages and processes)*" is marked as *'o'* in EventJava, because EventJava only supports Java language. The cell "*Explicit naming and grouping in ECL*" is marked as *'o'*, because EventJava only provides a fixed set of operators to compose and group events.

The cell "*Event declaration*" is marked as *'o'* in EScala, because in case of implicit events, EScala supports a fixed join point model. The cell "*Implicit invocation*" is marked as *'o'*, because in EScala in addition to implicit invocation, event publication is supported as well.

3.8 Conclusion

The existing runtime enforcement frameworks fall short in providing natural representation for the concepts of interest. This is mainly because the specification languages of these systems adopt the linguistic elements of their underlying implementation languages. The problem is amplified, since the current implementation languages fall short in representing the natural characteristics of runtime enforcement concepts. These characteristics are: a) transient nature of runtime enforcement concepts; b) open ended-ness of the kinds of elements in specifications; and c) no strict hierarchy among specifications.

To naturally support these features, the computational model Event Composition Model is introduced informally. This model is justified briefly against the desired features.

Event Composition Model	OO	AspectJ	AspectJ Extensions	Compose*	AWED	EOS	AspectWerkz	C#	EventJava	EScala	Ptolemy
Event declaration	-	-	+	-	-	-	-	+	+	o	+
Implicit invocation	-	+	o	+	+	+	+	-	-	o	-
Declarative ECL	-	+	+	+	+	+	+	-	-	+	+
Variable event composition strategies	-	-	-	o	-	-	-	-	-	-	-
Selecting events (from multiple languages and processes)	-	-	-	o	o	-	-	-	o	-	-
Explicit naming and grouping in ECL	-	-	o	-	o	o	-	-	o	-	-
Separation of interface from Implementations In event modules	o	o	o	+	-	+	+	-	-	-	-
Domain specific reactor types	-	-	-	+	-	-	-	-	-	-	-
Publishing events of implementation	+	o	-	-	-	+	+	+	+	+	+

Table 3.2: Implementing Event Composition Model in the existing programming languages

Finally, object-oriented languages, aspect-oriented languages, and the languages that support implicit invocation are evaluated to determine if they are eligible to implement Event Composition Model effectively. This assessment shows that none of the languages satisfy the requirements completely, although aspect-oriented languages are somewhat more suitable than the others.

Chapter 4

EventReactor: an Implementation of Event Composition Model

Chapter 3 provides a computation model, termed as Event Composition Model, to enable a natural representation of runtime enforcement concepts. Different programming languages are evaluated for their suitability in implementing the computation model. The evaluation reveals that none of the existing programming languages can implement the model as it is desired. According to the evaluation provided in Chapter 3, Compose* provides promising features in implementing Event Composition Model. However, considerable extensions to Compose* are required if it is to be adopted to implement Event Composition Model.

This chapter introduces the EventReactor language and compiler, as the successor of Compose*. EventReactor provides dedicated linguistic elements to define events, event modules, reactors, reactor types and reactor chains. It makes use of the Prolog language as its event composition language. EventReactor supports four kinds of primitive events that are identifiable in program code. It provides an API to programmers to declare so-called user-defined events.

The EventReactor language provides an API to programmers to define reactor types. It also provides dedicated operators to compose event modules with each other and to specify the constraints among them.

In the EventReactor language, the specifications are defined independently from any programming language, and the compiler of EventReactor facilitates generating code for Java, C and .Net languages. Reactors and reactor chains are parameterizable, and are defined separately from event modules. This increases the reusability of event modules and reactors.

EventReactor can be used as an underlying language for the specification languages of runtime enforcement frameworks, or it can be utilized to directly implement runtime enforcement techniques.

Section 4.1 explains the EventReactor language, and Section 4.2 discusses the compiler of the EventReactor language. The support of the EventReactor for predefined events is illustrated by means of an example in Section 4.3. Section 4.4 evaluates EventReactor with respect to the criteria provided in Chapter 3. A discussion about the similarities and differences between EventReactor and Compose* is given in Section 4.5. Future work and conclusion are outlined in Sections 4.6 and 4.7, respectively[1].

4.1 The EventReactor Language

The following subsections explain specification of events, event modules, reactor chains and reactor types in the EventReactor language.

4.1.1 Specification of Events

The EventReactor language supports a predefined set of events, which correspond to the following state changes: a) before invocation of methods; b) after invocation of methods; c) after invocation and immediately before execution of methods; and d) after execution of methods, which have terminated normally[2]. These are considered as predefined because the compiler of the EventReactor language identifies them in the program code and defines them in the language.

The language provides an API to programmers to define new sorts of events, which are termed as user-defined events.

The EventReactor language makes use of the Prolog language to select the events that are defined in the language. The predefined events are declared as Prolog facts by the compiler of the EventReactor language. The user-defined events must be declared as well. The definition of events provides the necessary information about the events and their publishers, so that it is possible to select them by the Prolog language.

Listing 4.1 shows a list of Prolog facts that are used to declare predefined events.

[1] An earlier version of this chapter is published in [14].

[2] Some aspect-oriented languages [53] can identify the states after execution of methods, which are not terminated normally. In our work, we only focus on the normal termination of an execution.

The expression `isBeforeInvocation` represented in line 1 is used to define an event that corresponds to the state change before an invocation of a method.

The argument `<event-name>` specifies the unique name of the event in the language. The unique name is assigned by the compiler of the EventReactor language.

The argument `<method-id>` specifies the unique identifier of the method of interest in the program code.

The character `'.'` in Prolog represents the termination of a fact.

Likewise, the Prolog expressions `isAfterInvocation`, `isBeforeExecution` and `isAfterExecution` are used to declare other sorts of predefined events.

The expression `isMethodWithName` represented in line 6 is used to provide necessary information about a method. The argument `<method-id>` represents the unique identifier of the method, which is assigned to by the compiler of the EventReactor language. This can be, for example, the signature of the method. The argument `<method-name>` represents the name of the method in the program code.

The expression `isClassWithName` represented in line 8 is used to provide necessary information about a class that is defined in the program code. The argument `<class-id>` represents the unique identifier of the class, which is assigned to by the compiler of the EventReactor language. The argument `<class-name>` represents the name of the class in the program code.

The expression `isDefinedIn` represented in line 10 specifies a relation among the program elements. In this example, it specifies that a specified method is defined within a specified class.

Other program elements, such as packages and interfaces, and the relations among the elements are also defined as Prolog facts. In [40], a list of Prolog expressions that exist in Compose* and are also reused by the EventReactor language is provided.

```
1  'isBeforeInvocation' '(' <event−name> ',' <method−id> ').'
2  'isAfterInvocation' '(' <event−name> ',' <method−id> ').'
3  'isBeforeExecution' '(' <event−name> ',' <method−id> ').'
4  'isAfterExecution' '(' <event−name> ',' <method−id> ').'
5
6  'isMethodWithName' '(' <method−id> ',' <method−name> ').'
7
8  'isClassWithName' '(' <class−id> ',' <class−name> ').'
9
10 'isDefinedIn' '(' <method−id> ',' <class−id> ').'
```

Listing 4.1: Prolog facts to represent predefined events

The Prolog facts representing user-defined events differ for each event. Chapter 7 provides examples of user-defined events that are defined to represent process-related events.

4.1.2 Specification of Event Modules

Listing 4.2 shows the structure of the linguistic construct `eventpackage`, which can be used to declare a group of event modules. The role of this construct is mainly grouping, like the package construct in Java. In addition, the programmers can define the constraints among the event modules declared in a single package.

Line 1 of Listing 4.2 indicates that every event package has a unique name in a given software system. Lines 2 and 3 are used to declare the events of interest, which is denoted by the keyword `selectors`.

The EventReactor language makes use of the Prolog language to select the events of interest that are defined in the language. Line 3 shows a Prolog variable and expression, represented by the non-terminals `<var-name>` and `<PrologExpression>`, respectively. The meaning of this statement is as follows. The Prolog expression selects a set of events, and stores the results in the Prolog variable `<var-name>`. Each set of selected events are associated with a unique name within its package. The name is specified by the non-terminal `<selector-name>`.

The statement in line 3 can be repeated for arbitrary times to declare groups of events.

Lines 5 to 9 show a declaration of an event module. The declaration includes binding a set of reactors to a set of events. In the following, this is explained in more detail.

The non-terminal `<eventmodule-name>` indicates that every event module must have a unique name in the enclosing package.

In line 6, the terminal `':='` can be interpreted as is *defined by*.

The non-terminal `<selector-name>,...` denotes a comma-separated group of selected events, which form the input interface of the event module.

The terminal `'<-'` is the keyword that indicates the binding of the specified reactors to the event module.

```
1  'eventpackage' <name> '{'
2   'selectors'
3      <selector−name> '=' '{' <var−name> '|' <PrologExpression> '};'
4
5   'eventmodules'
6      [<eventmodule−name> ':='] '{'<selector−name>,...'}'
7                                '<−'
8                                'perinstancethread' | 'perinstance' | 'perthread' | 'singleton'
9                                '{'<reactorchain−name>[<args>],...'};'
10
11  'constraints'
12      'precede' '(' <eventmodule−name> ',' <eventmodule−name> ');'
13      'override' '(' <eventmodule−name> ',' <eventmodule−name> ');'
14      'ignore' '(' <eventmodule−name> ',' <eventmodule−name> ');'
15 '}'
```

Listing 4.2: The structure of event packages

As it is explained in Section 3.2.2, four instantiation strategies are considered for event modules. These are specified in the specification of event modules via the keywords `perinstancethread`, `perinstance`, `perthread` and `singleton`.

The non-terminal `<reactorchain-name>` in line 9 refers to a reactor chain, which provides an implementation for the event module. Reactor chains can be parameterized. The non-terminal `<args>`, here shows the parameters that can be passed to the reactor chain. A comma-separated list of reactor chains can be bound to an event module.

The output interface of the event module is the union of the events that are published by the reactors attached to the event module. The list of these events is provided by the specification of reactor types, which is explained in Section 4.1.4.

In lines 11 to 14, by the help of the keyword `constraints`, the composition constraints among event modules are specified. Currently, the composition constraints `precede`, `override` and `ignore` are supported.

The constraint `precede(A, B)` specifies that if a selected event is a part of the input interface of two event modules `A` and `B`, the implementations of the event module A must react to the event first.

The constraint `override(A, B)` specifies that if a selected event is a part of the input interface of two event modules `A` and `B`, only the implementations of `A` must react to the event.

The constraint `ignore(A, B)` specifies that if a selected event is generated during the execution of reactors bound to the event module `B`, this event is ignored by the event module `A`.

4.1.3 Specification of Reactor Chains

Listing 4.3 shows the structure of a reactor chain in the EventReactor language.

The keyword `reactorchain` in line 1 indicates a declaration of a reactor chain, with the name `<rechain-name>`. A comma-separated list of optional parameters is specified via the expression `[<'?'param-name> | <'??'param-name>,...]`. Either the character `?` or characters `??` must proceed the name of the parameters. These indicate a single-valued and a multiple-valued parameter, respectively.

Each reactor chain may define two categories of attributes, which are denoted by the keywords `internals` and `externals`. The keyword `internals` defines the objects that are instantiated for each instance of a reactor chain. The keyword `externals` defines the objects that are instantiated *outside* the reactor chain. The externals may be referred to in different parts of the program, and be shared between the multiple instances of a reactor chain.

As shown in lines 2 and 7, internal or external attributes can be instantiated from a type as `<var-name> : <type>`. In case of an external attribute, it is also possible to specify the name of a method that returns the external attribute. This is shown in line 7 as `['='<method-name> '('[<arg-values>]')']`.

Every reactor chain declares a set of reactors, which are denoted by the keyword `reactors`. Each reactor is declared for the purpose of implementing one or more event modules. This is shown between the lines 9 and 13.

The expression `<reactor-name> : <type>` in line 10 shows an instantiation of a reactor from its type.

As shown by line 11, each reactor may specify its events of interest, which is specified with the keyword `event`, followed by a comma-separated list of events.

As shown by lines 12 to 15, a reactor may have a set of its own parameters, which can be used to initialize the reactor. The initialization of a reactor may be used for various purposes, for example to specify the behavior of the reactor with the help of its domain specific language.

If there are multiple reactors defined in a reactor chain, they are sequentially composed with each other using the operator `';'` as indicated in line 16. This means

an event goes through a chain of reactors, from the first specified one to the last one in the reactor chain.

```
'reactorchain' <rechain−name> [<'?'param−name> | <'??'param−name>,...] '{'
  'internals'
    <var−name> ':' <type> ';'

  'externals'
    <var−name> ':' <type> =
        ['='<method−name> '(' [<arg−values>] ')' ] ';'

  'reactors'
    <reactor−name> ':' <type>
            [ '=' '(' 'event' '==' '[' <events>,...']' ')' ]
            {
                <reactorparam−name> '=' <value>;
                ...
            }
    ';'
}
```

Listing 4.3: The structure of reactor chains

4.1.4 Specification of Reactor Types

Each reactor type has a **behavioral specification** that provides the following information about the reactor type: a) the name of the reactor type, b) whether it is read-only or read-write, c) the so-called action class that implements the functionality of the reactor type, and d) the name of events that can be published by the type. The structure of behavioral specifications is provided in Listing 4.4.

```
'@ReactorTypeDef' '(' 'name' '=' <type−name> ',' 'effect' '=' <read−only | read−write> ','
                      'implementation' '=' <actionclass−name> ',' 'events' '=' <event,...> ')'
```

Listing 4.4: The structure of behavioral specifications of reactor types

Listing 4.5 shows an excerpt of an action class. Here, the class `ReactorAction` is the base class for action classes, which is provided by the EventReactor language. The method `execute` implements the functionality of processing an event, for example, the functionality to enforce a property. Terminating the execution of software, invoking a method, and preventing the execution of a method are example actions that can be carried out by reactor types.

The event that must be processed is provided as an argument to the method. The execution context of the reactor is also provided as an argument to the method `execute`. The execution context contains, for example, information about the parameters of the reactor and the instance of the corresponding reactor chain.

For each reactor event whose name is specified in the behavioral specification of the reactor name, a method with the same name must be defined in the class. Invocations to such methods are considered as reactor events. The method `publish_an_event` in Listing 4.5 is an example.

```
1  public class anAction extends ReactorAction{
2
3     @Override
4     public void execute(RTEvent event, ReactorExecutionContext ctx) {
5           ...
6           publish_an_event();
7           ...
8     }
9
10    public Object publish_an_event() {
11       ...
12    }
13 }
```

Listing 4.5: An example implementation of a reactor type

4.2 The Compiler of the EventReactor Language

Figure 4.1 provides a global overview of the compiler of the EventReactor language.[3] The EventReactor language does not incorporate the elements of the underling programming language; and its compiler supports the Java, C and .Net languages.

The modules *Event Recorder*, *Event Module Recorder* and *Analyzer* of the compiler are language-independent, and the modules *Type Harvester*, *Code Generator* and *Weaver* must be implemented for each supported programming language.

[3]This section mainly focuses on a high-level design of the compiler, rather than the implementation details.

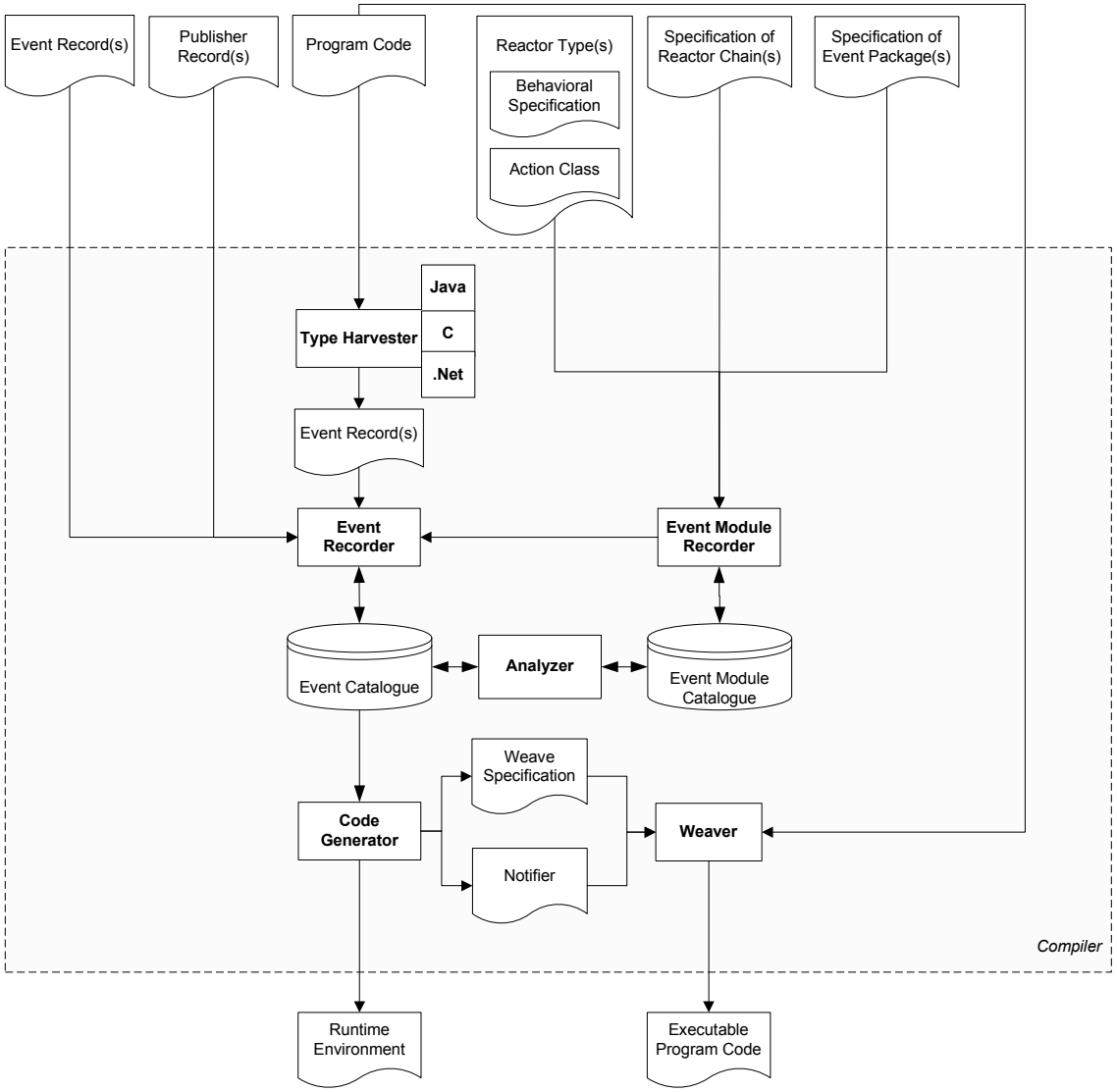

Figure 4.1: An overall view of the EventReactor compiler

4.2.1 Input and Output of the Compiler

As shown at the top of the Figure 4.1, the compiler receives the following input: *Event Record(s)*, *Publisher Record(s)*, *Program Code*, *Reactor Type(s)*, *Specification of Reactor Chain(s)*, and *Specification of Event Packages(s)*.

The language supports a predefined set of events, whose information is extracted by the compiler of the EventReactor language from program. The language and compiler are also extendable with user-defined events. These are defined via the input *Event Record(s)*, which are Prolog facts with the necessary static contextual information about the event. This facilitates the selection of the events via Prolog queries.

The necessary information about the publishers of the user-defined events is provided as a set of Prolog facts via *Publisher Record(s)*.

The input *Program Code* represents the program publishing user-defined events and/or the program from which the list of predefined events must be extracted. As it is explained before, the compiler generates a set of Prolog facts to represent the predefined events.

The language provides an API to define new types of reactors. Each type of reactor is represented by a *Behavioral Specification* and an *Action Class* that implements the functionality of the reactor type.

Specification of Event Packages(s) and *Specification of Reactor Chain(s)* contain the information about the event modules and reactor chains, respectively.

As its output, the compiler creates the runtime environment of the program and modifies *Program Code* such that it can announce events to the runtime environment. These are shown as *Runtime Environment* and *Executable Program Code* in Figure 4.1, respectively.

4.2.2 Event Catalogue

The Prolog facts defined by *Event Record(s)* and *Publisher Record(s)* are stored in *Event Catalogue*. The module *Event Recorder* is the interface to this catalogue.

As shown in Figure 4.1, the module *Type Harvester* accepts *Program Code* as input. Since *Program Code* can be expressed in different languages, *Type Harvester* converts the input to a common internal representation. This representation contains, for example, the following information: a) the static structure of the program code, in terms of the classes defined in the program; b) the interfaces that are implemented

by each class; c) the methods and attributes defined in each class; and d) The methods that are invoked by the classes.

For each invocation on a method, two event records are created: a) one for the event that represents the state change before the invocation; b) one for the event that represents the state change after the invocation.

In addition, for each method defined in a class, two event records are created: a) one for the event that represents the state change after the invocation and immediately before the execution of the method; b) one for the event that represents the state change after the execution of the method, which terminates normally.

Type Harvester assigns unique identifiers to the identified events, and to the code segments (e.g. method and class) in *Program Code* from which the events are to be published. This information is also maintained in the event records.

The module *Event Recorder* receives these records from *Type Harvester* and stores them in *Event Catalogue*.

4.2.3 Event Module Catalogue

The module *Event Module Recorder* accepts *Reactor Type(s)*, *Specification of Reactor Chain(s)* and *Specification of Event Package(s)* as input, and performs the following:

It stores a reference to *Reactor Type(s)* in *Event Module Catalogue*.

It checks whether the reactor types which are used in *Specification of Reactor Chain(s)* exist in *Event Module Catalogue*.

It parses *Specification of Event Packages(s)*, and extracts the following information: list of specified event modules, the specified events of interest that form the input interface of event modules, the reactor chains that are attached to event modules and the instantiation strategy of event modules. These are stored in *Event Module Catalogue* to be used at runtime for instantiation of event modules.

It extracts the list of reactor events by parsing behavioral specification of reactor types, specifications of event modules and specifications of reactor chains. For each event, an event record is created, which defines a set of Prolog facts expressing the static contextual information of the event. This information specifies the name of the event, the name of the corresponding reactor type, the name of reactors declared from the reactor type, the name of reactor chains in which these reactors

are declared, and the event modules to which the reactor chains are bound. The Prolog facts are provided to *Event Recorder* to be stored in *Event Catalogue*

4.2.4 Analysis and Checking

Although events occur at runtime, the event records stored in *Event Catalogue* facilitate performing various checks statically on the specifications. These checks are performed by the module *Analyzer*.

The module *Analyzer* evaluates the specified Prolog queries stored in *Event Module Catalogue* against the Prolog facts stored in *Event Catalogue*, and selects the event records that satisfies the queries.

Analyzer keeps a link between an event record that is stored in *Event Catalogue*, the selectors that match the event record, and the corresponding event modules. This information is maintained in *Event Module Catalogue*. If a selector is not used in the declaration of any event module, *Analyzer* reports a warning to the programmers.

Afterwards, *Analyzer* performs the following checks on the specifications of event packages:

- If no event record is selected by the specified Prolog queries, *Analyzer* reports a warning indicating that the specified events are not defined in the language.
- If a selected event record is referred to by multiple event modules, *Analyzer* checks whether the constraint *precede* is specified. If not, *Analyzer* reports a warning to indicate that the reactor chains are to be executed in a random order.
- If a selected event record is referred to by two event modules A and B, *Analyzer* checks whether the constraint *ignore (A,B)* is specified. If so, *Analyzer* tags the reactors that are bound to A as conditional. As a result, the corresponding reactors ignore the events that are published during the execution of the reactors bound to B.
- If the constraint *override(A, B)* is specified, it checks whether a selected event record is referred to by both event modules A and B. If it is the case, *Analyzer* tags the reactors bound to B as conditional. As a result, the corresponding reactors do not process the events.

Analyzer stores the above-mentioned changes in the specification of reactors in *Event Module Catalogue*.

In addition, *Analyzer* checks the side-effect freeness of the read-only reactor types.

4.2.5 Code Generation

The module *Code Generator* creates the executable program and the runtime environment of the EventReactor language for the program. These are shown in Figure 4.1 as *Executable Program Code* and *Runtime Environment*, respectively. The code generation is performed as follows:

To be able to locate the code segments that publish the predefined events to the corresponding event modules, *Code Generator* retrieves the corresponding event records from *Event Catalogue*. The information about the location of the code segments is provided as *Weave Specification*.

Code Generator creates the module *Notifier(s)* as part of the runtime environment of the EventReactor language, which informs the runtime environment of the occurrence of the specified predefined event.

The generated *Weave Specification* and *Program Code* are input to the module *Weaver*, which inserts invocations to *Notifier(s)* in specified places in *Program Code*.

4.3 An Illustrative Example: File Access Control

This section makes use of an example to illustrate a possible use of the EventReactor language. The example focuses of the support of the EventReactor language for predefined events.

Assume that a runtime enforcement system has to be designed to ensure that a file is utilized correctly. The correct usage of the file must comply with the usage protocol defined for the file. The usage protocol of a file is defined as follows: First the files must be opened, then zero or more read and write operations can be performed on the file, and finally the file must be closed. If the utilization of the file does not satisfy the usage protocol, two recovery actions have to be carried out: (1) Reporting of the error to the user; (2) Preventing the access to the file.

In the following, the specifications of event packages and reactor chains along with a discussion about the dedicated reactor types that are implemented for this example are explained.

Listing 4.6 shows the specification of the event package for checking the correct usage of a file. As line 1 shows, the name of the package is `file_usage`.

Lines 2 to 25 select the events of interest. Lines 3 to 7 select the event corresponding to the state before the file is opened.

```
1  eventpackage file_usage{
2      selectors
3          open_event = {E |
4                              isBeforeExecution(E, M),
5                              isMethodWithName(M,'open'),
6                              isClassWithName(C,'File'),
7                              isDefinedIn(M, C)};
8
9          close_event = {E |
10                             isBeforeExecution(E, M),
11                             isMethodWithName(M,'close'),
12                             isClassWithName(C,'File'),
13                             isDefinedIn(M, C)};
14
15         read_event = {E |
16                             isBeforeExecution(E, M),
17                             isMethodWithName(M,'read'),
18                             isClassWithName(C,'File'),
19                             isDefinedIn(M, C)};
20
21         write_event = {E |
22                             isBeforeExecution(E, M),
23                             isMethodWithName(M,'write'),
24                             isClassWithName(C,'File'),
25                             isDefinedIn(M, C)};
26
27     eventmodules
28         file_eventmodule := {open_event, close_event, read_event, write_event} <−
29                             perinstancethread
30                             {regularexpression_REChain('
31                                 (open_event (read_event | write_event)∗ close_event)∗')};
32 }
```

Listing 4.6: A specification of event module for the usage protocol of a file

In line 4, the Prolog expression `isBeforeExecution (E, M)` selects the events `E`, which correspond to the state after the invocation and immediately before the execution of the methods `M`.

The expression `isMethodWithName (M, 'open')` in line 5 selects the methods whose names match the string `'open'`.

The Prolog expression `isClassWithName(C,'File')` in line 6 selects all the classes whose names match the string `'File'`.

The Prolog expression `isDefinedIn(M, C)` selects the methods `M` that are defined in classes `C`.

The character `','` between the Prolog expressions is a conjunction operator.

As a result of these queries, all the events that correspond to the state changes after the invocation and immediately before the execution of the method `open` defined in class `File` are selected.

The assumption here is that the file is represented by class `File` in the program code, and the events correspond to the methods that are to be executed on the instances of this class. Such events have the same name as their corresponding methods.

Lines 9 to 25 select the other three events of interest, which are named as `close_event`, `read_event` and `write_event`.

Lines 28 to 31 define the event module `file_eventmodule` as follows. It groups the events selected by `open_event`, `close_event`, `read_event`, and `write_event`; and binds the reactor chain `regularexpression_REChain` as its implementation, and initializes the reactor chain with a parameter.

The parameter is expressed in a domain specific language, which is a regular expression formula. Here, `(open_event (read_event| write_event)* close_event)*` indicates that a correct usage of a file starts with the event `open_event`, followed by zero or more times `read_event` or `write_event`, and finally terminated by the event `close_event`.

Since the keyword `perinstancethread` is used in the declaration of the event module, for each individual instance of class `File` and each thread of execution in which the methods of `File` are invoked and executed, a separate instance of the event module is created. This facilitates verifying the usage of each `File` object separately.

Listing 4.7 shows the definition of the reactor chain `regularexpression_REChain` with one initialization parameter, which represents a regular expression formula.

```
1 reactorchain regularexpression_REChain(?regformula){
2     reactors
3         regexp_REC: RegularExpression {
4             reactor.formula = ?regformula;
5         };
6 }
```

Listing 4.7: A specification of the reactor chain for the usage protocol of a file

The reactor chain declares the reactor `regexp_REC` of type `RegularExpression`. We implemented the type `RegularExpression` for this thesis. It is a read-only reactor type, which receives a regular expression formula as its input argument, and checks a selected event against the formula. If the formula is violated, it publishes the event `violated`. The runtime behavior of `RegularExpression` is explained in Section 5.5.1.

In Line 4 of Listing 4.7, the parameter `?regformula` is assigned to the attribute `formula` that is defined by type `RegularExpression`.

Assume that the method `open` is invoked on an instance of class `File`. According to the statements in lines 3 to 7 of Listing 4.6, the events that correspond to the state changes after the invocation and immediately before the execution of the method `open` are selected and named as `open_event`. These events form the input interface of the event module `file_eventmodule`. Therefore, this event module is notified and the reactor chain that is bound to it receives the events.

The reactor `regexp_REC` checks these events against its regular expression formula. If the events violate the formula, `regexp_REC` publishes the event `violated` with a reference to the event `open_event`.

Listing 4.8 defines an event module to implement the recovery actions for the example. Line 1 defines the event package `file_recoveryactions`.

Lines 3 to 7 show three Prolog statements. The events that are selected by these statements are named as `verification_event`. The meaning of these statements is described as follows. The event `violated`, which is published by the event module named `file_eventmodule` is selected.

```
1  eventpackage file_recoveryactions{
2      selectors
3          verification_event = {E |
4                                  isEventWithName(E, 'violated'),
5                                  isEventModuleWithName(EM, '*.file_eventmodule'),
6                                  isPublishedBy(E, EM)
7                               };
8
9      eventmodules
10         recovery_eventmodule := {verification_event} <- singleton {recovery_actions_REChain};
11 }
```

Listing 4.8: A specification of event module for the recovery actions

Line 10 defines the event module `recovery_eventmodule`, which groups the events selected by `verification_event`, and as its implementation specifies the reactor

chain `recovery_actions_REChain`. Since the keyword `singleton` is used, at runtime only one instance of `recovery_eventmodule` is created to handle the event `violated` published by all instances of the event module `file_eventmodule`.

Listing 4.9 shows the definition of the reactor chain `recovery_actions_REChain`. Lines 3 to 5 define the reactor `logger_REC` of type `Log` that is implemented for this thesis. It is a read-only reactor type, which reports a message on the screen when the selected events occur. The message to be reported on the screen is passed to the reactor via the parameter `info`.

Line 7 defines the reactor `forcereturn_REC` of type `ForceReturn` that is also implemented for this thesis. It is a read-write reactor type, which prevents a method to be executed by returning the flow of execution to the caller of the method.

The reactor `forcereturn_REC` processes the event `violated` published by the reactor *regexp_REC* defined in Listing 4.7. This event keeps a reference to the event whose occurrence caused the regular expression fail. Through this reference, `forcereturn_REC` retrieves information about the method whose execution must be prevented, and prevents the execution of the method. The details of the runtime behavior `ForceReturn` is explained in Section 5.5.2.

Since the reactors `logger_REC` and `forcereturn_REC` are composed with each other using the operator `';'` in line 6, they form a chain such that `logger_REC` processes the selected events first.

```
1 reactorchain recovery_actions_REChain{
2     reactors
3         logger_REC: Log {
4             reactor.info = 'the file is accessed incorrectly';
5         }
6     ;
7     forcereturn_REC: ForceReturn;
8 }
```

Listing 4.9: A specification of reactor chain for the recovery actions

4.4 Evaluation of the EventReactor Language

This section evaluates the EventReactor language with respect to the following requirements, which are defined in Section 3.3 as the typical characteristics of a language that can be used in implementing Event Composition Model.

- Declaration of arbitrary state changes as an event: In the EventReactor language, user-defined events can be expressed as a set of Prolog facts, which are stored in *Event Catalogue*. This facilitates defining arbitrary state changes as events.
- Implicit invocation on reactors: An interface is provided for the publishers to announce events. For the predefined kinds of events, *Notifiers* are generated for this matter. Upon the occurrence of an event, the runtime environment informs the corresponding event modules and the reactor chains. This is an implementation of implicit invocation.
- Selection of any event that is defined: EventReactor makes use of Prolog as its event composition language. Prolog enables the definition of a rich set of expressions to select the defined events.
- A declarative event composition language: Since Prolog is a declarative language, it is possible to perform various sorts of analysis. For example, the specifications *precede*, *ignore* and *override* can be analyzed as defined in the previous section.
- Explicit naming of the group of events that are selected: Event modules are named, and dedicated Prolog expressions are available to select and refer to them.
- Definition of a set of reactor types: The language is extendable with new reactor types.
- Declaration of one or more instances from a reactor type: Within a reactor chain, it is possible to declare one or more reactors from a reactor type.
- Binding a reactor instance to an event module: Reactor chains are defined separately from the event modules. The EventReactor language provides a dedicated operator for binding.
- Modularized reactor types: The separation of reactor types from event modules increases the modularity of event modules.
- Programmable reactor types: It is possible to define reactor types that are parameterized by specifications in their domain-specific languages. The reactor type *RegularExpression*, which accepts a regular expression formula as its parameter, is an example.
- Publishing events by reactor types: Reactor types may publish new events. Dedicated Prolog expressions are provided to select these events and group them in the definition of event modules.

- Variety of composition strategies: Since reactor types may publish new events, which can be selected, definition of hierarchy of event modules is supported. The event modules residing at the higher levels of the hierarchy can compose the event modules residing at the lower levels. Dedicated reactor types can be provided to express various composition strategies among the events that are published by the lower level event modules.

4.5 Similarities and Differences with Compose*

Since the EventReactor language has been inspired from Compose*, in this section we elaborate on the similarities and differences between these two languages.

The similarities are as follows:

- Reactor types resemble filter types in Compose*.
- Both reactor types and filter types may be parameterized by specifications in their domain-specific languages.
- Both languages are open-ended in the definition of these types.
- Reactor chains resemble filter modules in Compose*.
- Both reactor chains and filter modules may declare *internal* and *external* attributes.
- Reactor chains and filter modules are parameterizable.
- Both languages make use of the Prolog language for the selection elements of interest.
- Both languages support language- and platform-independent specifications.

The differences between these two languages are summarized as follows:

- The composition language of Compose* is limited to selecting classes and superimposing filters on them. The filters only process incoming and outgoing messages to/from objects.

  In contrary to Compose*, the event composition language of EventReactor can select arbitrary sorts of events, provided that they are known in the language. For this reason, the following features are provided:

  - There are dedicated Prolog expressions to select individual and/or groups of declared events.
  - Since events come to life at runtime, it is facilitated to detect the events and inform their occurrence to the corresponding event modules. This is in contrary to Compose* in which Prolog expressions and superimpositions are evaluated at compile time.
  - Events are first-class entities, in the sense that they can be selected and passed as arguments to reactor chains.

- The composition language of Compose* is limited to superimposing individual instances of filters (modules) on individual objects only. The event composition language of EventReactor, however, facilitates grouping events that are published by single and/or multiple objects. These objects can be implemented in different languages and distributed across multiple processes.

- Reactor types can publish events; this cannot be realized by filters in Compose*.

- In Compose* it is not possible to declare filter types as read-only.

- Superimpositions in Compose* are not named; therefore, it is not possible to refer to them. In EventReactor, however, event modules are named, and dedicated Prolog expressions are provided to select them.

- EventReactor introduces four instantiation strategies for event modules.

- EventReactor provides three composition constraints *precede*, *ignore* and *override*.

4.6 Future Work

In the previous sections, the compiler of the EventReactor language was explained. Implementing a full-fledge compiler for a language like EventReactor is a labor-intensive task. Due to the limited resources, the following strategy was followed to demonstrate the feasibility of the compiler:

The similarities between EventReactor and Compose*, which are discussed in the previous section, enabled us to reuse and extend the modules of the Compose* compiler in implementing the EventReactor language. Here is a brief summary of the modules that are reused/extended:

- The module *Type Harvester* of Compose* is extended to extract the predefined events in program code.
- The module *Event Module Recorder* partially reuses the features provided by the Compose* compiler to parse the specifications and stores them in a shared data storage.
- The module *Code Generator* is partially reused to create the executable programs, and to generate *Notifiers*.
- The module *Weaver* is taken as it is.
- The high-level design of the compiler of the EventReactor language is similar to the Compose* compiler, where shared data storages are used for exchanging information among the modules.

As a proof of concept several distinguishing features of the EventReactor compiler have been implemented and published in various papers [63, 61]. For example in [63], we discuss an implementation of the compiler that facilitates selecting events from a group of objects that are implemented in different languages. The possibility to publish events from reactor (filter) types and forming a hierarchy of specifications are also explained in [63]. In [61], we explain the feature of the compiler that facilitates selecting events from a group of objects distributed across multiple processes.

Some of the features of the compiler are currently being investigated. Ensuring the side-effect freeness of reactor types is an example. In the following, we focus on how the side-effect freeness of reactors can be checked effectively.

For each reactor type that is specified as read-only in its behavioral specification, *Analyzer* checks that the reactor type do not have functional side-effects on other reactors and/or on the program. There are several research works [52, 3, 95, 96] in the aspect-oriented area that categorize aspects based on their side-effects on the program, and analyzes aspects to infer the category to which they belong. The so-called *spectative* or *observer* aspects [52, 3] are the ones that do not have functional side-effects on the program. We consider the read-only reactor types similar to the spectative aspects.

There are data-flow analysis tools [3, 96] based on the Soot framework [87], which analyze an aspect to infer whether it is spectative. We consider reusing these tools in the module *Analyzer* to ensure that the reactors specified as read-only are actually side-effect free. Adopted from [3, 96], the following analysis is to be performed for each reactor type that is specified as read-only.

1. Identifying the variables, including the parameters of the reactor, that are bound to the variables defined in *Program Code*, to the internal or external attributes of reactor chains, or to the attributes of the events.

2. Checking that none of the variables identified in the previous step appears on the left-hand side of an assignment in the reactor type. The checking is also performed for all methods that are invoked from within the reactor type, to ensure that the identified variables are not also modified from within these methods.

3. Checking that the code of the reactor type terminates. Although generally an undecidable problem, syntactic special cases such as loop-freeness can be checked.

4. Checking that the the code of reactor type does not disable independent underlying operations. This is applicable to the aspect-oriented languages in which an aspect can change the structure of base software by introducing new fields, methods and classes to the software. The details can be found in [96].

5. Checking that the processing of an event is resumed at each point where the execution was interrupted by reactors. In addition, checking that the enabling conditions for the underlying operations have not been affected by the reactors. Thus there cannot be exceptions thrown in the reactor types that lead to abnormal termination or do not resume the underlying execution at the point from which it was interrupted.

4.7 Conclusion

This chapter introduces the EventReactor language, which implements Event Composition Model. The chapter explains in detail the syntax of the language and its compiler. By means of an example, the suitability of the EventReactor language for implementing Event Composition Model is illustrated.

The chapter provides detailed evaluation of EventReactor with respect to the requirements mentioned in Chapter 3. The chapter discusses similarities and differences between EventReactor and Compose* in detail.

Chapter 5

Design of the Runtime Environment of EventReactor

This chapter discusses the runtime environment of the EventReactor language. Section 5.1 outlines the requirements that must be addressed in the design of the language. Section 5.2 explains the data structures employed by the EventReactor language to represent events and event modules. The runtime behavior of the EventReactor language is discussed in Section 5.3, and is illustrated by means of an example in Section 5.4.

The EventReactor language is extendable with new types of reactors, and each type has its dedicated runtime behavior. Section 5.5 explains the runtime behavior of two reactor types *RegularExpression* and *ForceReturn*, which are used in implementing a runtime enforcement example. Finally, Section 5.6 outlines the conclusion.

5.1 Requirements for the Implementation of the Runtime Environment

To be able to identify the requirements in the design of the EventReactor language, we assume that a typical execution flow has the following steps:

1. Publishers of the events of interest are created: For example, if an event of interest corresponds to the state after the execution of a method by an object, the object is regarded as the publisher of the event, which is created at some point during the execution of the program.

2. An event of interest occurs.
3. The event is detected and published to the runtime environment in a synchronous manner.
4. The runtime environment identifies the corresponding event modules and the reactor chains that are bound to them.
5. The runtime environment chooses a suitable instantiation strategy for the identified event modules and reactor chains, based on their specifications. Afterwards, it instantiates the event modules and the reactor chains accordingly.
6. The runtime environment delegates the detected event to the corresponding reactor chains.
7. The runtime environment applies the specified constraints to the reactors in processing events.
8. The event is processed by reactors.
9. While processing the event, the reactors may publish new events, or may provide the results of their operation as state variables.
10. The flow of execution eventually returns to the publisher of the event, which is blocked due to its synchronous communication with the runtime environment to publish the event. The necessary information about the operations performed on the event is also returned to the publisher.

A closer look at the above-mentioned steps and the features of the EventReactor language, which are explained in Chapter 4, helps us to observe the following facts:

- The events that are published can be either predefined or user-defined. Accordingly, we classify the publishers as predefined or user-defined.
- Published events must be defined in the language.
- Various publishers may be correlated with each other; hence, the events that are published by them may also be correlated with each other. The correlated events are grouped as event modules.
- Individual instances of event modules and their corresponding reactor chains may be created for each group of correlated publishers that announce events in the same thread of execution. This is termed as **per-instance-thread** instantiation strategy.

- Individual instances of event modules and their corresponding reactor chains may be created for each group of correlated publishers, regardless of the thread of execution in which they announce the events. This is termed as **per-instance** instantiation strategy.
- Individual instances of event modules and their corresponding reactor chains may be created for all correlated publishers, depending on the thread of execution in which they announce events. This is termed as **per-thread** instantiation strategy.
- A singleton instance of event modules and their corresponding reactor chains may be created, shared among all the correlated publishers and the threads of execution. This is termed as **singleton** instantiation strategy.
- Correlated publishers may be distributed across multiple processes.
- Correlated publishers may be implemented in different languages.
- Multiple instances of event modules may exist to process an event.
- Three sorts of constraints can be applied to reactors: *precede*, *ignore* and *override*.

Based on the above-mentioned flow of execution and facts, we identify the following requirements to be addressed in the design of the runtime environment of the EventReactor language:

- Data structures are required to represent events at runtime. These must facilitate maintaining sufficient static and dynamic contextual information for events. The static contextual information helps to match events to the event records stored in *Event Catalogue* to ensure that the events are known in the language.
- Data structures and algorithms are required to keep track of groups of correlated publishers, which may be distributed across multiple processes and/or implemented in different languages.
- Data structures and algorithms are required to represent the instances of event modules and their corresponding reactor chains, and to bind them to the corresponding group of correlated publishers.
- Data indexing and retrieval mechanisms are required to identify the instances of event modules that must process an event.

- Standard interfaces are required to receive events published by various different kinds of publishers.

5.2 Data Structures

This section explains the data structures designed for the runtime environment of the EventReactor language, such that the requirements identified in the previous section are addressed. Relevant design alternatives are explained for each case, if any.

5.2.1 Representing Events at Runtime

Events are represented by the data structure named *RTEvent*, which has three fields: *StaticContext*, *DynamicContext* and *ReturnContext*.

The field *StaticContext* keeps the following two attributes to represent the static context of an event: a) *name* represents the name of the event; and b) *PrologFacts*, which as its name implies, represents a set of Prolog facts that are used to match a published event and declared events in *Event Catalogue*.

The field *DynamicContext* keeps a list of attributes representing the dynamic contextual information of the event. Each attribute is defined by a name, a type and a value. We consider three predefined attributes: a) *thread* represents the unique identifier of the thread of execution which causes the event; *process* represents the unique identifier of the process where the event occurs; and c) *publisher* represents the unique identifier of the publisher of the event.

In EventReactor, the predefined events correspond to the state changes before and after the invocation and the execution of methods. For the predefined events, we consider the following extra attributes to represent the dynamic context of an event : a) *caller* represents the object that invokes a method of interest. b) *callee* represents the object on which the method is invoked. c) *args* represents the arguments of the method; the return value of the method is also represented by an argument. d) *stacktrace* represents the stack trace[1] of the methods whose executions cause the invocation of the method.

Reactor events are published when a reactor processes an event. For the reactor events, we consider the following two predefined attributes: a) *innerEvent* repre-

[1]A stack trace is a report of the active stack frames at a certain point in time during the execution of a program.

sents the event whose processing causes a reactor event be published. b) *stacktrace* represents the stack trace of the methods whose executions cause a reactor event be published.

User-defined events may have their own dedicated attributes.

As a result of event processing, reactors may want to return information to the publisher of an event. The field *ReturnContext* keeps a list of attributes representing such information. These attribute varies for each kind of event.

For predefined events we consider the predefined attribute *flow* for the return context, which indicates the flow of execution after an event is processed. This attribute can have one of the following values: a) *Continue* means that the corresponding invocation and/or the execution of the method of a predefined event must proceed. b) *Return* means that the corresponding invocation and/or the execution of the method of a predefined event must not proceed. c) *Exit* means that the execution of the program must terminate.

5.2.2 Representing Event Modules and Publishers at Runtime

To keep track of groups of correlated publishers and their corresponding instances of event modules and reactor chains, the runtime environment of the EventReactor language creates a so-called **event module table** for each specified event module. In the following, the structure of event module tables is explained.

The Structure of Event Module Tables

Event module tables are implemented as relational database tables [80]. Each table has the name of the corresponding event module.

Each table has a column named *col_thread*, which keeps the unique identifier of a thread of execution.

For each reactor chain that is bound to the event module, there is a separate column named as *col_<reactorchainname>*, which stores a reference to an instance of the reactor chain.

For each set of selected events, which are identifiable by `selector-name` (see Listing 4.2), there is a separate column named *col_publisher_<selector-name>*. The column stores the unique identifier of the publishers of the selected events.

If an event module is specified to be instantiated as per-instance-thread, each row of the table will represent a separate instance of the event module. An instance is created for each combination of the correlated publishers participating in the same thread of execution.

If an event module is specified to be instantiated as per-instance, the column *col_thread* of the corresponding table will have the value *null*. Each row of the table will represent a separate instance of the event module for a group of correlated publishers, regardless of the threads of execution in which they publish events.

If an event module is specified to be instantiated as per-thread, the value *null* will be stored in the columns *col_publisher_<selector-name>* of the corresponding table. Each row of the table will represent a separate instance of the event module for each thread of execution, without distinguishing between the correlated publishers that participate in the thread.

If an event module is specified to be instantiated as singleton, the value *null* will be stored in the columns *col_publisher_<selector-name>* and *col_thread* of the corresponding table. The table will always have one row referring to a single instance of the event module.

By adopting a relational data structure for representing the correlations among publishers and the instances of event modules, we benefit from the common relational operations on tables.

Sharing of Event Module Tables

We consider the following alternatives in the sharing of event module tables at runtime:

- There is only one publisher for the selected events. In this case, a direct reference can be provided from the publisher to its corresponding event module tables. This reduces the look up time for the event modules.

- A group of correlated publishers executing in a single process publish the selected events. In this case, we consider the following implementation alternatives:

    1. The publishers share the tables through a common storage. Upon the occurrence of an event, the runtime environment of EventReactor looks up for the corresponding tables directly.

2. The publishers share the tables through a common storage. In contrast to the first case, each publisher has a direct access to the corresponding rows. This may reduce the required lookup up time.

   If a new set of correlated publishers is activated, or a new thread of execution starts, a new sets of rows will be created in the tables to represent the participation of the correlated publishers in the thread of execution. This implies that the references, which are provided to the publishers, must be updated accordingly.

3. A local copy of the table is provided to each publisher. When a new publisher is activated or a new thread of execution is started, the distributed tables must be synchronized to maintain the consistency. Similar to the previous case, synchronization of multiple tables causes additional overhead.

- There is a group of correlated publishers distributed across multiple processes: The three alternatives mentioned in the previous case can be also considered here. We identify the following challenges:

  1. If the tables are shared by multiple processes, a performance bottleneck [25] may occur.

  2. If a reference to the corresponding rows are provided to each publisher, updating these reference may impose considerable overhead due to the latency in the inter-process communications. The same problem may also appear if local copies of the tables are provided to publishers.

To be able to choose the best alternative, one needs to consider the actual timing characteristics of the systems being considered. In this thesis, we consider maintaining the event module tables in a shared data storage a practical solution. In the rest of the thesis, we use the term **table repository** to refer to such a data storage.

If all the specified events of interest are the predefined-ones, and all the publishers are executing in a single process, the table repository is also maintained in the same process. If publishers are distributed across multiple processes, the table repository is maintained in a dedicated process, separately from the application processes. If any of the specified events of interest is the user-defined one, the process in which the tables must be maintained is specified by programmers and its information is provided to the compiler.

5.3 Runtime Behavior

The runtime environment makes use of the Observer design pattern [34] to provide a standard interface for publishers. The publishers must register themselves for the interface and send instances of the data structure *RTEvent* to announce the events.

If a group of correlated publishers is implemented in the same language, the interface is also created in the same language. This eases the intercommunication between the publishers and the interface.

If a group of correlated publishers is implemented in different languages, the interface is created in the Java language. Publishers that are implemented in languages other than Java make use of Java-JNI [45] technique to communicate with the interface.

This section explains the runtime behavior in publishing and processing events. The behavior is explained for the case publishers are executing in a single process. A group of correlated publishers is distributed across multiple processes. The runtime behavior for the distributed case is explained in Chapter 6.

Derived from the scenario explained in Section 5.1, we consider the following significant cases at runtime.

5.3.1 Publishers of the events of interest are created

As it is explained in Section 4.2, for the predefined events, the compiler of the EventReactor language has identified the code segments where the events of interest are published, and inserted invocations to the module *Notifier* there.

In object-oriented programs, we consider objects as predefined publishers. Upon the creation of an object, which publishes an event, firstly, an instance of the module *Notifier* is created and is bound to the object. Secondly, *Notifier* assigns a unique identifier to the object. Thirdly, *Notifier* registers itself for the corresponding runtime interface. In this way, *Notifier* functions as a wrapper for the publisher object.

In non-object-oriented programs, we consider software files as predefined publishers to which the module *Notifier* is bound at compile time. *Notifier* assigns a unique identifier to each file to be used as the unique identifier of the publisher.

User-defined publishers become known when they are registered using the corresponding interface.

If the event is published by a reactor, an instance of *Notifier* is created and bound to the reactor.

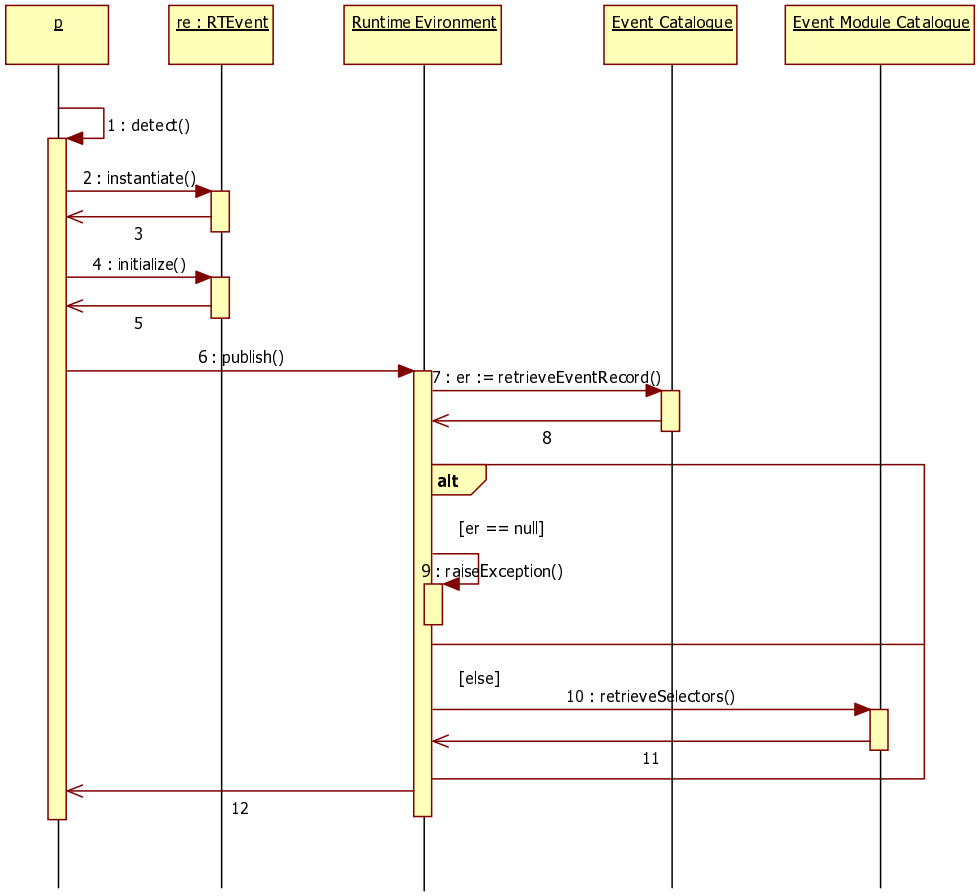

Figure 5.1: The sequence of actions to detect and publish an event

5.3.2 An event of interest is detected and published

Assume that an event is detected and published in a thread of execution. We represent the event, the thread of execution and the publisher as e, t and p, respectively.

Figure 5.1 makes use of a UML sequence diagram [68] to show the sequence of actions to detect and publish an event. First, p creates an instance of the data structure *RTEvent*, and initializes the necessary static and dynamic contextual information to represent e. This information, among others includes p and t as the unique identifier of the publisher and the unique identifier of the thread of execution, respectively.

The announcement of the event is performed by sending the instance of *RTEvent* to the corresponding interface provided by the runtime environment of the EventReactor language.

The runtime environment keeps a copy of *Event Catalogue*, to check if the published event is known in the language or not.

The runtime environment reads the static contextual information of the event, and converts it to a set of Prolog queries. It evaluates the queries against the facts stored in *Event Catalogue* to retrieve the event record that matches the event. If there is no such a record, the runtime environment raises an error. Otherwise from *Event Module Catalogue* the list of specified selectors that match the event record is retrieved.

5.3.3 Corresponding event modules are instantiated

The compiler of the EventReactor language keeps a link between the following items: the event record that is stored in *Event Catalogue*, the selectors that match the event record, and the corresponding event modules. This information is maintained in *Event Module Catalogue*. As an illustrative example, let us assume that <*selector-e*> matches the event record that is selected in the previous step.

The sequence of actions to identify the event modules that correspond to a specified selector is shown in Figure 5.2. The runtime environment keeps a copy of *Event Module Catalogue* to access the information about the specified selectors and event modules. It retrieves the list of event modules that refer to the selector <*selector-e*>. If there is no any event module, it ignores the event.

Let us also assume that <*eventmodule-e*> refers to <*selector-e*>. The runtime environment looks up for the table <*eventmodule-e*> in the table repository. The following possibilities are considered:

1. **There is no such a table**: This happens when none of the events of interest has been occurred so far; therefore, no event module was instantiated. In this case, the runtime environment creates a table and inserts a row in the table.

   If the event module is specified to be instantiated as per-instance-thread, the runtime environment retrieves the unique identifier p of the publisher from the corresponding instance of *RTEvent*. The value p is then inserted in the column *col_publisher_*<*selector-e*> of the newly added row. The unique identifier of the corresponding thread t is also retrieved from the instance of *RTEvent*, and inserted in the column *col_thread* of the same row.

   If the event module is specified to be instantiated as per-instance, only p is inserted in the column *col_publisher_*<*selector-e*> of the row.

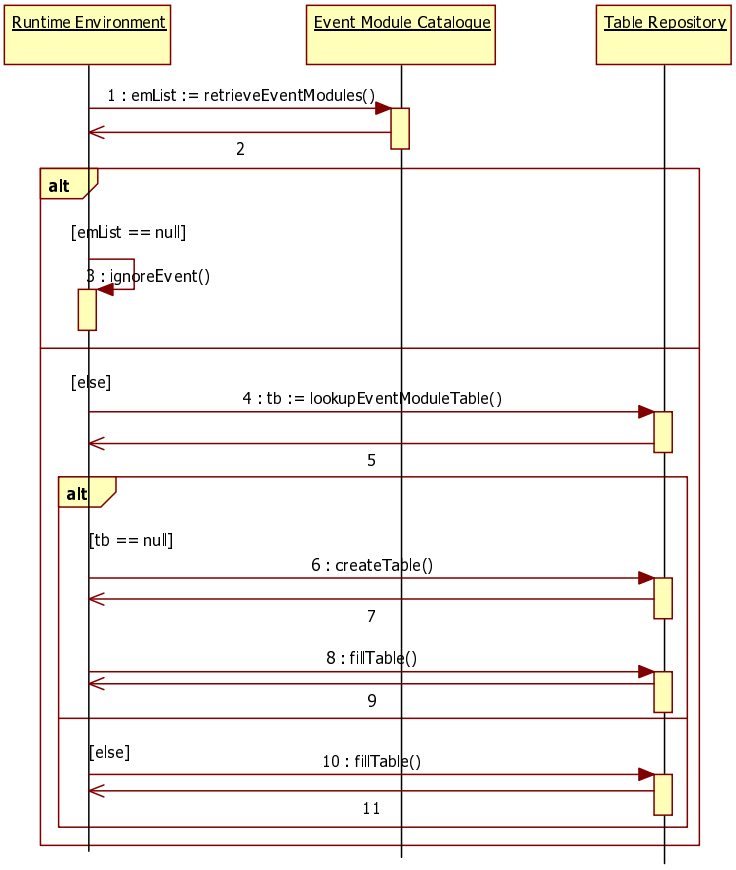

Figure 5.2: The sequence of actions to identify event modules

If the event module is specified to be instantiated per-thread, only t is inserted in the column *col_thread* of the row.

If the event module is specified to be instantiated as singleton, the value *null* is inserted in (a) all the columns corresponding to the publishers, and (b) the column that represents the thread of execution.

From *Event Module Catalogue*, the information about the reactor chains that are bound to the event module is retrieved. The reactor chains and their enclosed reactors are instantiated, and references to the instantiated reactor chains are inserted in the corresponding columns in the table.

2. **If the table contains at least one row**: The following cases are considered:

   (a) If the event module is specified to be instantiated as singleton, the detected event is processed by the reactor chains. These are instantiated as defined in the previous case.

   (b) If the event module is specified to be instantiated as per-instance thread, the runtime environment selects all the rows whose column *col_publisher_<selector-e>* contains p and whose column *col_thread* contains t.

   If any row is selected, the event is processed by the corresponding instances of the reactor chains.

   If no row is selected, the following cases are considered:

      i. There are rows whose column *col_publisher_<selector-e>* contains *null*, and whose column *col_thread* contains t. This happens when some other correlated publishers are already participated the in t. In this case, the runtime environment replaces the value *null* with p. As a result, the publisher will participate in the thread of execution along with the other correlated publishers.

      ii. There are rows whose column *col_thread* contains t, but the column *col_publisher_<selector-e>* does not contain *null*: This case happens when multiple publishers announce a defined event and p announces this event in t for the first time.
      In this case, a new set of rows is added to the table, for representing the participation of p together with the other correlated publishers in t. The value p is inserted in the column *col_publisher_<selector-e>* of the newly added rows, and t is inserted in the column *col_thread*. The unique identifier of the correlated publishers are inserted in their corresponding column. New instances of reactor chains are created for these newly added rows, and the event is processed by the reactor chains.

iii. There is no row whose column *col_publisher_<selector-e>* contains p, or whose column *col_thread* contains t: This case happens when p is the first publisher among the correlated publishers that announces an event of interest in the thread of execution t.
In this case, a new row is added to the table, with p in its column *col_publisher_<selector-e>*, and t in its column *col_thread*. The value *null* is inserted in the columns corresponding to the other publishers. New instances of the reactor chains are created for this new row, and the event is processed by the reactor chains.

(c) If the event module is specified to be instantiated as per-instance, the runtime environment selects all the rows whose column *col_publisher_<selector-e>* contains p.

If any row is selected, the event is processed by the corresponding instances of the reactor chains.

If no row is selected, the cases are similar to the ones of 2.b.i, 2.b.ii and 2.b.iii are considered, except that the unique identifier of t is not taken into the account.

(d) If the event module is specified to be instantiated as per-thread, the runtime environment selects all the rows whose column *col_thread* contains t.

If a row is selected, the event is processed by the corresponding instances of the reactor chains.

If no row is selected, a new row is added to the table, with t in its *col_thread*. A new instance of the reactor chains is created for this new row, and the event is processed by the reactor chains.

5.3.4 An event is processed by the corresponding reactor chains

After the corresponding event modules and reactor chains are instantiated, the event represented by an instance of *RTEvent* is processed by the reactor chains. This is performed in the following steps:

1. Based on the information available in *Event Module Catalogue*, the order of reactor chains in processing the event is determined. The order can be defined by the help of two operators: *precede* in the specification of event packages, or *';'* in the specification of reactor chains.

2. In processing the event, the operators *ignore* and *override* are considered into account, if relevant. Starting from the first reactor in the list, if any of these operators are utilized (see Section 4.2.4), the event processing is continued by the next reactor in the list.

3. If a reactor is selected, the event is checked against the specification of events of interest (see Section 4.1.3) that is defined for the reactor, if any. If the event matches the specification, the reactor will process it. Otherwise, the event is passed to the next reactor in the list, and the processing continues from the step 2.

4. After the termination of the execution of the last reactor, the flow-of-control is returned to the publisher of the event. Reactors may return the result of the executions through the field *ReturnContext* of the corresponding instance of *RTEvent*.

During the processing of an event, a reactor may publish new events. Similar to other events, instances of *RTEvent* are used to represent the necessary static and dynamic information. Announcement to the corresponding reactor is carried out by *Notifier* that is bound to the publishing reactor.

5.3.5 An event or an event module is passed as argument

In the EventReactor language, it is possible to pass an event or an instance of an event module as an argument to reactor chains. When the runtime environment is informed of an event or when it instantiates an event module, it retrieves the list of corresponding reactor chains from *Event Module Catalogue*, and passes the event or the instance of the event module as argument. In the current version of the EventReactor language, only the event modules that are instantiated as singleton can be passed as arguments.

5.3.6 A publisher no longer exists

If a publisher no longer exists, using the unique identifier of the publisher, the corresponding instances of reactors are retrieved from the event module tables and informed. When the notification is handled, the corresponding rows in the event module tables are tagged as *inactive*. Every reactor type may respond to such notification differently.

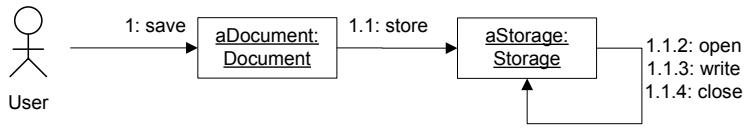

Figure 5.3: The sequence of events to store a document

5.3.7 A thread of execution is terminated

In this case, none of the events of interest occur in a terminated thread. With the help of the unique identifier of a terminated thread, all the corresponding rows are retrieved from all event module tables. Afterwards, a notification to the corresponding instances of reactors are sent. Every reactor type may respond to such notification differently. When the notification is handled, the corresponding rows in the event module tables are tagged as *inactive*.

5.4 Illustration of Runtime Behavior

Assume that there is a document-editing software, with two core modules *Document* and *Storage*, which provide services to edit a document and to save its contents, respectively. Assume that it is required to verify at runtime that a request to save a document by the user eventually results in storing the document content on the file system.

Figure 5.3 shows a UML collaboration diagram [68], which depicts the sequence of events that handles this request. We assume that each event corresponds to the state change after the invocation and immediately before the execution of a method on modules *Document* and *Storage*.

If the events that handle the user's request do not occur as specified in the figure, the execution of the corresponding method must be prevented, and an error message must be logged.

The following subsections define the specifications of event modules and reactor chains for this example, and explain the runtime behavior of the application.

5.4.1 Specifications of Event Modules and Reactor Chains

Listing 5.1 defines the event package `user_request`. Lines 3 to 7 select the event `save` that corresponds to the state immediately before the execution of the method

`save` on instances of class `Document`. The selected event is represented by the variable `save_event`.

Similarly, lines 9 to 31 select the events that correspond to the states immediately before the execution of the methods `store`, `open`, `write` and `close` on the instances of class `Storage`.

Lines 34 to 40 define the event module `document_storage_eventmodule`, by grouping the selected events and binding the reactor chain `regularexpression_REChain` to them. The argument of the reactor chain specifies the acceptable sequence of events as a regular expression formula.

```
1  eventpackage user_request{
2      selectors
3          save_event = {E |
4                                  isBeforeExecution(E, M),
5                                  isMethodWithName(M,'save'),
6                                  isClassWithName(C,'Document'),
7                                  isDefinedIn(M, C)};
8
9          store_event = {E |
10                                 isBeforeExecution(E, M),
11                                 isMethodWithName(M,'store'),
12                                 isClassWithName(C,'Storage'),
13                                 isDefinedIn(M, C)};
14
15         open_event = {E |
16                                 isBeforeExecution(E, M),
17                                 isMethodWithName(M,'open'),
18                                 isClassWithName(C,'Storage'),
19                                 isDefinedIn(M, C)};
20
21         close_event = {E |
22                                 isBeforeExecution(E, M),
23                                 isMethodWithName(M,'close'),
24                                 isClassWithName(C,'Storage'),
25                                 isDefinedIn(M, C)};
26
27         write_event = {E |
28                                 isBeforeExecution(E, M),
29                                 isMethodWithName(M,'write'),
30                                 isClassWithName(C,'Storage'),
31                                 isDefinedIn(M, C)};
32
33     eventmodules
34         document_eventmodule := {save_event, store_event, open_event,
35                                    write_event, close_event}
36                                  <-
37                                  perinstancethread
38                                  {regularexpression_REChain('
39                                (save_event store_event open_event write_event+ close_event)*'
40                                  )} ;
41 }
```

Listing 5.1: A specification of event module for the document-editing software

The definition of `regularexpression_REChain` is presented in Listing 5.2. Here, the reactor `regexp_REC` is defined of type `RegularExpression`, which accepts a regular expression in its `formula` attribute.

```
reactorchain regularexpression_REChain(?regformula){
    reactors
        regexp_REC: RegularExpression {
            reactor.formula = ?regformula;
        };
}
```

Listing 5.2: A specification of reactor chain for storing a document

Listing 5.3 shows the specification of an event module for the recovery actions. The event `violated`, which is published from the event module `document_eventmodule` defined in Listing 5.1, is selected and forms the input interface of the event module `recovery_eventmodule`. The reactor chain `recovery_actions_REChain` is bound to this event module, and `singleton` is specified as the instantiation strategy of the event module.

```
eventpackage file_recoveryactions{
    selectors
        verification_event = {E |
                                isEventWithName(E, 'violated'),
                                isEventModuleWithName(EM,
                                  'user_request.document_eventmodule'),
                                isPublishedBy(E, EM)};

    eventmodules
        recovery_eventmodule := {verification_event}
                                <-
                                singleton {recovery_actions_REChain};
}
```

Listing 5.3: A specification of event module for the recovery actions

Listing 5.4 shows the implementation of `recovery_actions_REChain`. Here, the reactors `logger_REC` and `forcereturn_REC` are defined of types `Log` and `ForceReturn`, respectively. These types are explained in Chapter 4.

```
reactorchain recovery_actions_REChain{
    reactors
        logger_REC: Log {
            reactor.info = 'the sequence of events is incorrect.'; }
        ;
        forcereturn_REC: ForceReturn;
}
```

Listing 5.4: A specification of reactor chain for the recovery actions

5.4.2 Runtime Behavior

Table 5.1 shows the structure of the table *user_request.document_eventmodule*, which is created for the event module *document_eventmodule* (see Listing 5.1).

Table 5.1: The structure of the table *user_request.document_eventmodule*

col_thread: STRING	col_publisher_save_event: STRING	col_publisher_store_event: STRING	col_publisher_open_event: STRING	col_publisher_close_event: STRING	col_publisher_write_event: STRING	col_regularexpression_REChain: OBJECT

The column *col_thread* maintains the unique identifier of the thread of execution in which the corresponding event has occurred. Since in Listing 5.1, five sets of events are selected, there are five columns to store the unique identifier of the publishers of the events. As defined in Listing 5.1, only one reactor chain is bound to the event module `document_eventmodule`, whose instance is stored in the column `col_regularexpression_REChain`.

Figure 5.4 shows the runtime view of EventReactor for the illustrative example. In the figure, *aDocument* and *aStorage* represent the instances of class *Document* and *Storage*, respectively.

As the name indicates *Runtime Environment* represents the runtime environment of the program, which accesses *Event Module Catalogue*, *Event Catalogue*, and *Table Repository*.

Assume that the following scenario happens at runtime:

1. **A predefined publisher is created**: This matches the case explained in Section 5.3.1.

   In an object-oriented program, objects are considered as predefined (potential) publishers of events. Assume for example, the object *aDocument* is created at runtime. According to Listing 5.1, the event *save_event* can be published from instances of class *Document*. Therefore, upon the instantiation of *aDocument*, an instance of *Notifier* is created and bound to it. This instance is named as *Notifier(D)* in Figure 5.4. Similarly, when the object *aStorage* is created, *Notifier(S)* is created and bound to it.

2. **The event *save* is published by *aDocument***: This matches the cases explained in Sections 5.3.2 and 5.3.3.

   Assume that during the execution thread *t*, the method *save* is invoked on the object *aDocument*. According to the specification in Listing 5.1, the event

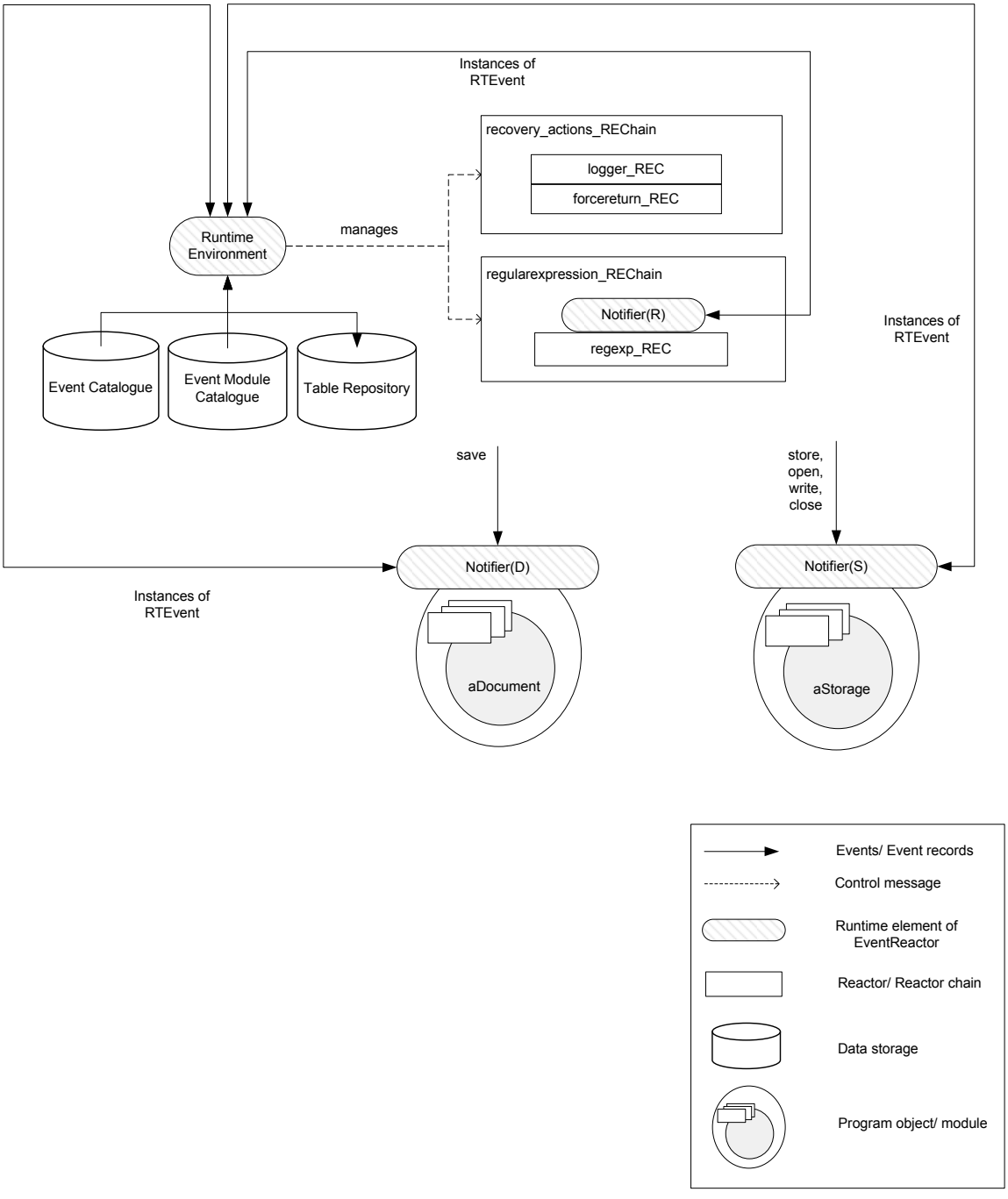

Figure 5.4: The runtime view of the document-editing software

corresponding to the state change after the invocation and immediately before the execution of *save* must be published.

As it is explained in Section 5.3.2, *Notifier(D)* creates an instance of class *RTEvent*. For the static contextual information, *'save'* is stored as the name of the event. The Prolog facts

isBeforeExecution(e1, m1).
isMethodWithName(m1,save).
isClassWithName(c1,Document).
isDefinedIn(m1,c1).

are kept to identify the state to which the event correspond. The literals *e1*, *m1* and *c1* represent the unique identifiers that the module *Type Harvester* of the EventReactor compiler assigned to the event, to the corresponding method, and to the class in which the method is defined.

For the dynamic contextual information, *Notifier(D)* specifies *aDocument* as the publisher and the active stack frame as the stack trace of the event. Other reflective information about the method, e.g. its call and return arguments are also kept as a part of the dynamic contextual information. The value t is used as the unique identifier of the thread of execution where the event has occurred.

An instance of *RTEvent* is published by *Notifier(D)*.

As it is explained in Section 5.3.3, the static contextual information about the event is read and converted into a set of Prolog queries. The queries are then evaluated against the facts that have been stored in *Event Catalogue*. If an event record matches, it is retrieved. In this example scenario we assume that there is a matching event in *Event Catalogue*.

A list of event modules that refer to the event are then retrieved. In our example, it is *document_eventmodule*. Afterwards, it looks up for the table *user_request.document_eventmodule* in *Table Repository*.

Since *save* is the first event of interest that is published, the corresponding table has not been created yet. This matches the case 1 in Section 5.3.3. Here, first a table is created and a new row is inserted. Within this row, the unique identifier of *aDocument* and the thread of execution t are stored in the columns *col_publisher_save_event*, and *col_thread*, respectively. The value *null* is inserted in the columns *col_publisher_store_event*, *col_publisher_open_event*, *col_publisher_close_event* and *col_publisher_write_event*.

From *Event Module Catalogue*, the information about the reactor chains, which are bound to the event module is retrieved. In the illustrative example, this is

regularexpression_REChain. After the initialization of the reactors, a reference to these is inserted in the column *col_regularexpression_REChain*.

The content of the table *user_request.document_eventmodule* is as follows.

col_thread: STRING	col_publisher_ save_event: STRING	col_publisher_ store_event: STRING	col_publisher_ open_event: STRING	col_publisher_ close_event: STRING	col_publisher_ write_event: STRING	col_regularexpression _REChain: OBJECT
t	ID of aDocument	null	null	null	null	Ref(1) to regularexpression_ REChain

In the illustrative example, according to the specification in Listing 5.3, the events published from within the event module *document_eventmodule* are selected for the purpose of recovery. To inform the occurrence of these events, *Notifier(R)* is created and bound to the instance of the reactor type *regexp_REC*.

After the instantiation, the event *save* is processed by the reactor *regexp_REC*.

3. **The event *store* is published by *aStorage* in the thread *t***: Similar to the event *save*, an instance of *RTEvent* is created to pass the necessary information about this event.

   At this stage, the table *document_eventmodule*, which contains a single row has been created and instantiated already with the following data: The columns *col_thread* and *col_publisher_store_event* contain the values *t* and *null*, respectively.

   As it is explained by the case 2.b.i in Section 5.3.3, the value *null* is replaced with the unique identifier of *aStorage*, and the event is provided to the reactor chain instance which is kept in the column *col_regularexpression_REChain*.

   The content of the table *user_request.document_eventmodule* is as follows.

col_thread: STRING	col_publisher_ save_event: STRING	col_publisher_ store_event: STRING	col_publisher_ open_event: STRING	col_publisher_ close_event: STRING	col_publisher_ write_event: STRING	col_regularexpression _REChain: OBJECT
t	ID of aDocument	ID of aStorage	null	null	null	Ref(1) to regularexpression_ REChain

4. **The event *close* is published by *aStorage* in the thread *t***: Similar to the previous case, the value *null* in the column *col_publisher_close_event* is replaced with the unique identifier of *aStorage*, and the event is provided to the corresponding instance of the reactor chain..

   The content of the table *user_request.document_eventmodule* is as follows.

col_thread: STRING	col_publisher_save_event: STRING	col_publisher_store_event: STRING	col_publisher_open_event: STRING	col_publisher_close_event: STRING	col_publisher_write_event: STRING	col_regularexpression_REChain: OBJECT
t	ID of aDocument	ID of aStorage	null	ID of aStorage	null	Ref(1) to regularexpression_REChain

The occurrence of the event *close* violates the regular expression specified in Listing 5.1. As a result, the reactor *regexp_REC* publishes the event *violated*. This is accomplished by the corresponding *Notifier(R)* through creating and instantiating an instance of *RTEvent*.

For the static contextual information, *Notifier(R)* initializes the name of the event as *'violated'* and the publisher as *'user_request.document_eventmodule. regularexpression_REChain.regexp REC'*.

The dynamic contextual information is initialized as follows: the unique identifier of *regexp_REC* is stored as the identifier of the publisher. The attribute *innerEvent* is set to *RTEvent*, which corresponds to the event *close*. As it is explained in Section 5.2.1, *innerEvent* is a predefined attribute, which refers to the event that caused a reactor event be published. The value *t* is set as the unique identifier of the current thread of execution.

RTEvent is published by *Notifier(R)*.

5. **The event *violated* is processed by *ForceReturn***: When the reactor *forcereturn_REC* receives the event *violated*, it retrieves the information about the event that caused *violated* be published, which is the event *close* in this case. Afterwards, the reactor retrieves the information about the corresponding method in the program, which is the method *close* of *aStorage*.

   Since *close* is a predefined event, it has the attribute *flow* in its *ReturnContext*. The reactor sets the value *Return* for this attribute. This indicates that the invocation and/or the execution of the method must not proceed.

   When the control returns to *Notifier(S)*, it checks the attribute *flow*, and accordingly it prevents the execution of the method *close* to proceed.

6. **Another instance of class *Document* is created and publishes the event *save***: There can be multiple instances of a class that publish events of interest in the same or different threads of executions. Assume the object *aDocument'* is another instance of *Document*, and in the thread of execution *t* the method *save* is invoked on *aDocument'*.

   Similar to the previous cases, upon the creation of *aDocument'* an instance of *Notifier* is created and is bound to it. *Notifier* announces the occurrence of the event *save*.

There is already a table named *user_request.document_eventmodule* in the table repository. There is already one row in the table, whose column *col_thread* contains *t*. But the column *col_publisher_save_event* of this row does nor contain the unique identifier of *aDocument'*. This is similar to the case 2.b.ii explained in Section 5.3.3.

Here, a new row is added to the table. The unique identifier of *aDocument'* and *t* are inserted in the corresponding columns of the newly added row. The unique identifier of the other publishers are inserted in the corresponding columns. A new instance of the reactor chain *regularexpression_REChain* is created and its reference is inserted in the corresponding column of this row too. The new row indicates that when *aDocument'* starts participating in the thread *t*, *aStorage* has already been participating in the thread *t*, and has already published the events *store* and *close*.

The content of the table *user_request.document_eventmodule* after this scenario is as follows.

col_thread: STRING	col_publisher_save_event: STRING	col_publisher_store_event: STRING	col_publisher_open_event: STRING	col_publisher_close_event: STRING	col_publisher_write_event: STRING	col_regularexpression_REChain: OBJECT
t	ID of aDocument	ID of aStorage	null	ID of aStorage	null	Ref(1) to regularexpression_REChain
t	ID of aDocument'	ID of aStorage	null	ID of aStorage	null	Ref(2) to regularexpression_REChain

7. **The event *save* is published by *aDocument* in the thread *t'***: Assume that *t'* is a new thread of execution in which *save* is published. This matches the case 2.b.iii in Section 5.3.3.

   A new row is added to the table *user_request.document_eventmodule*. The value *t'* and the unique identifier of *aDocument* are inserted in the corresponding columns. The value *null* is inserted in the columns corresponding to the other publishers. A new instance of the reactor chain *regularexpression_REChain* is created and its reference is inserted in the column *col_regularexpression_REChain* of this row.

   The new row indicates that *t'* is a new thread of execution and so far only *aDocument* has participated in this thread.

   The content of the table *user_request.document_eventmodule* after this scenario is as follows.

col_thread: STRING	col_publisher_save_event: STRING	col_publisher_store_event: STRING	col_publisher_open_event: STRING	col_publisher_close_event: STRING	col_publisher_write_event: STRING	col_regularexpression_REChain: OBJECT
t	ID of aDocument	ID of aStorage	null	ID of aStorage	null	Ref(1) to regularexpression_REChain
t	ID of aDocument'	ID of aStorage	null	ID of aStorage	null	Ref(2) to regularexpression_REChain
t'	ID of aDocument	null	null	null	null	Ref(3) to regularexpression_REChain

5.5 Runtime Behavior of Reactor Types

The runtime behavior of a reactor type is defined by the implementation of that type. This section describes types *RegularExpression* and *ForceReturn*, which are illustrative examples for implementing dedicated reactor types.

5.5.1 The Runtime Behavior of *RegularExpression*

The type *RegularExpression* wraps the 3rd-party tool [2] that accepts a regular expression and translates it to a deterministic finite state automaton according to the algorithm discussed in [43]. The 3rd-party tool provides an API to programmers through which the transition function δ can be invoked on the created automaton. The reactor type *RegularExpression* makes use of this API to verify an event against a regular expression formula. It publishes various reactor events according to the result of verification.

The automaton is defined as $A := <Q, \Sigma, \delta, q_0, F, \Theta>$, where:

- Q is a finite set of states.
- Σ is a finite set of events, specified by the regular expression predicate, which must be selected for verification.
- δ is a transition function that takes an event $e \in \Sigma$ and a *state* $\in$ Q as its input, and returns a *state* $\in$ Q.
- $q_0 \in$ Q is the start state.
- F $\subseteq$ Q is the set of final (or accepting) states.
- $\Theta \in$ Q is the trap state, which is reached by all events that do not satisfy the regular expression predicate.

Algorithm 1 shows the runtime behavior of the reactor r of the type *RegularExpression* in verifying the event e.

Algorithm 1 The algorithm for verifying the event e by the reactor r from the type *RegularExpression*

```
1: if m indicates the termination of the thread t or e indicates the destruction of
   a publisher then
2:    if currentState ∈ F then
3:       return
4:    end if
5: end if
6: targetState ← δ(e, currentState)
7: define result as a reactor event
8: result.innerEvent← e
9: result.publisher← r.uniqueID
10: if currentState = q0 then
11:    result.name← 'started'
12:    publish result
13: end if
14: currentState ← targetState
15: if targetState = Θ then
16:    result.name← 'violated'
17:    publish result
18:    return
19: end if
20: if targetState ∈ F then
21:    result.name← 'validated'
22:    publish result
23:    return
24: end if
25: result.name← 'intermediate'
26: publish result
```

The algorithm defines the variables *currentState*, *targetState* and *result*, which respectively keep the current state of the automaton, the target state of the transition caused by the verification of e, and the event that is generated as the result of the verification. The algorithm assumes that in the beginning, *currentState* has the value q_0.

In lines 1 to 5, if e indicates a systemic event, i.e. the termination of an execution thread or the destruction of a publisher, and if the automaton is in a final state, the

event is ignored and the algorithm terminates. This means that e does not influence the results of the verification, because the regular expression predicate is already satisfied. Otherwise, line 6 verifies e by invoking the transition function δ. The result is returned in the variable *targetState*.

Lines 7 to 9 define the reactor event *result*, set the event e as its inner event, and set the unique identifier of the reactor r as the publisher of it.

In lines 10 to 13, if *currentState* equals to q_0, the reactor event is named *started* indicating that the verification is started, and the event is published. Afterwards *currentState* is updated with *targetState*.

In lines 15 to 19, if *targetState* is the trap state, the reactor event is named *violated*, is published and the processing terminates.

Likewise, in lines 20 to 24 the reactor event is named *validated*, and is published if *targetState* is a final state.

In lines 25 and 26, the reactor event is named *intermediate*, and is published to indicate that the final result of the verification is not determined yet.

According to this algorithm, if the verification fails, the automaton remains in the trap state, and any other event causes the reactor event *violated* be published.

In certain applications, it may be necessary to reset the instance of *RegularExpression* to its initial state. This can be programmed by accessing the variable *currentState* which is defined in the dynamic context of the reactor event.

5.5.2 The Runtime Behavior of *ForceReturn*

As it is discussed in Section 5.2.1, at runtime an event is represented by an instance of *RTEvent*, which defines various attributes for the event. For the predefined events, the attribute *flow* is defined as the return context of the event. It indicates the flow of execution after an event is processed by a reactor.

As shown in Algorithm 1, reactors may publish new events while processing an event. The attribute *innerEvent* stores a reference to the original event. This is shown in line 8 of Algorithm 1. It is possible to access the chain of events that are causally published after each other through this attribute. The reactor type *ForceReturn* assumes that such a chain of events is initiated by a predefined event.

When a reactor of type *ForceReturn* receives an event, it iterates over the chain of events until it reaches the instance of *RTEvent* that represents a predefined event.

The reactor type sets the value *Return* for the attribute *flow* of this event. This indicates that the invocation and/or the execution of the method must not proceed.

When the control returns to *Notifier*, which is bound to the publisher of the predefined event, it reads the attribute *flow* and accordingly prevents the the execution of the corresponding method.

5.6 Conclusion

This chapter presents the requirements that must be fulfilled in the design and the implementation of the runtime environment of the EventReactor language. It describes the data structures that are used to represent events, groups of correlated publishers, and instances of event modules and their corresponding reactor chains. Based on this information, the chapter explains the runtime behavior of the EventReactor language, and illustrates it by means of an example. Along this line, the reactor types *RegularExpression* and *ForceReturn* are explained.

This chapter explains several alternatives in designing the runtime environment of the EventReactor language. Each alternative has its advantages and disadvantages. A precise evaluation requires a detailed analysis of the specific characteristics of the implementation context. As future work, we consider carrying out more practical experiments along this line.

Chapter 6

Multiplicity of Processes and Implementation Languages

Software may adopt various process structures during its life time. This may happen, for example, to fulfill scalability and performance requirements. The changes to the process structure may be manually applied or may be supported by tools [91].

Software may be also implemented using various programming languages. There is a considerable number of software systems implemented in multiple languages. Typically, for example, modules which directly interact with hardware, such as device drivers, are implemented in the C language. The Java language is generally adopted in implementing application modules and user interfaces. Throughout the lifetime of software, a module may be replaced with another one that is implemented in a different language.

To support software systems with various process structures and/or implementation languages, this chapter claims that a runtime enforcement framework must fulfill the following requirements: a) its specification languages must abstract from the process structure through distribution transparency (i.e. process and location transparency) at least to the level that is provided by the middleware. b) Its specification languages and compiler must facilitate specifying and enforcing end-to-end behavior of software. Here, the term end-to-end behavior refers to a sequence of execution traces from the first initiating event until the final event; the execution traces are causally dependent on each other and may belong to different processes. c) Its compiler must facilitate automatic generation of enforcement components for arbitrary process structures. d) Its specification language must be transparent from the im-

plementation languages of software; and e) Its compile must facilitate generating runtime enforcement components so that multiple-language software is supported.

Fulfilling these requirements increases the applicability of a runtime enforcement framework to software systems with various process structures and implementation languages. In addition, it increases the reusability of specifications for such software.

In this chapter, Section 6.1 elaborates on the shortcomings of the existing runtime enforcement frameworks in dealing with the variety of process structures and implementation languages. The chapter discusses the degree to which these requirements are addressed by the EventReactor language and its compiler. Fulfilling all the requirements without making any compromise is a challenging task.

This chapter elaborates on two approaches. The first solution, which is explained in Section 6.2, is to offer process transparency and end-to-end verification without supporting multiple implementation languages. In this case, the implementation of the EventReactor language is based the Java-RMI technology, and therefore it only supports the Java platform. Section 6.3 explains the second solution, which assumes a more relaxed form of distribution transparency, nevertheless offers implementation language transparency. Section 6.4 outlines the conclusion and future work.[1]

6.1 Problem Statement

It becomes a commodity to develop distributed software on top of middleware. Distributed Java software that makes use of Java-RMI for inter-process communication is an example. The behavior of distributed software is accomplished by a sequence of events exchanged among objects (modules) running in one or more processes. The objects (modules) may be implemented in the same or different languages.

This section first provides an illustrative example, which is used to discuss the shortcomings of the existing runtime enforcement frameworks. Second, it presents a set of requirements for runtime enforcement frameworks in case multiple-process and/or multiple-language software has to be supported. Finally, it elaborates on the shortcomings of the existing runtime enforcement frameworks in fulfilling these requirements.

[1] Earlier versions of this chapter are published in [61, 63, 62].

6.1.1 An Illustrative Example

Consider for example that there is a distributed implementation of the document-editing software which was introduced in the previous chapter. Figure 6.1 shows a UML collaboration diagram, which depicts the sequence of events that handles user's request to save a document.

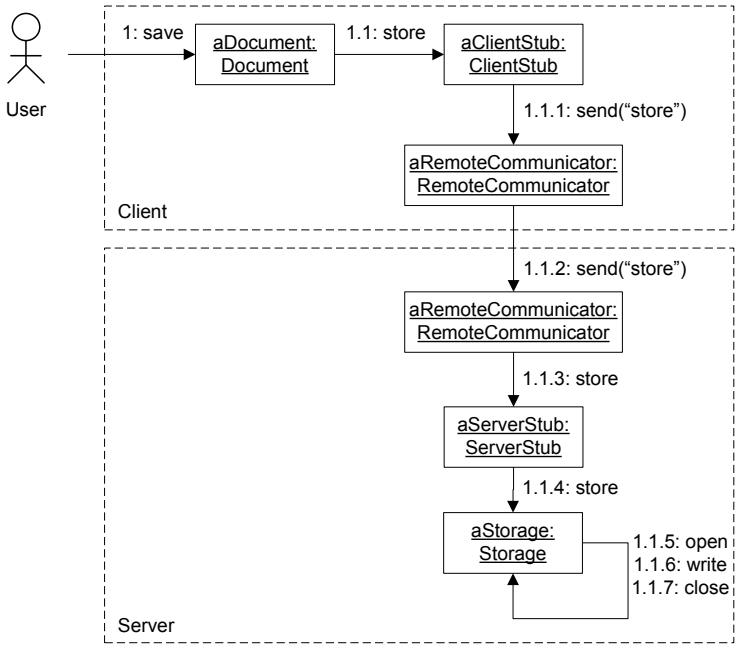

Figure 6.1: An example causally-dependent sequence of events

Starting from the left, the request to save a document by the user results in a set of causally dependent occurrence of events; first *save* occurs on the object *aDocument* in the process *Client*, subsequently *store* on the objects *aClientStub*, *aRemoteCommunicator*, *aServerStub*, and finally, on *aStorage* in the process *Server*.

In this implementation, the sequence of events is distributed across two processes. To facilitate distribution and inter-process communication, a middleware is utilized. Here, *aClientStub*, *aServerStub* and *aRemoteCommunicator* are representing the middleware objects.

6.1.2 Requirements for a Runtime Enforcement Framework

To effectively apply runtime enforcement to software systems with various process structures and implementation languages, the thesis claims that a runtime enforcement framework must fulfill the following requirements:

1. **Distribution transparency in specifications**: As middleware facilitates the development of distributed software by abstracting the distribution details away, at least the same level of distribution-transparency should be provided in specifications. Consequently, a) the specification task is eased, because the details of process structure are not included in the specifications; b) if the process structure of software changes without changing its algorithmic behavior, the specification may be preserved; c) there is a conformance between the development of distributed software and its specification in terms of transparency.

2. **End-to-end enforcement**: To ensure the correctness of a distributed sequence of events, it is necessary to specify and verify a sequence of application events that occur in one causal thread of execution, from the first initiating event until the occurrence of the final event. Here, the term causal thread refers to a thread of execution that starts from a process and spans across multiple processes.

3. **Automatic generation of enforcement modules for arbitrary process structures**: To apply runtime enforcement to multiple-process software, in principle, one needs to specify and verify the events exchanged in the application part and the middleware part of software integrally. Nowadays, middleware becomes more and more reliable. By considering middleware as a reliable component, to ensure the correctness of software behavior, it is sufficient to focus on the sequence of application events.

   For example, in Figure 6.1, the events 1.1 to 1.1.3 can be excluded from the verification. However, it is still necessary to consider the causal dependency of application events in *Server* to the application events in *Client*, in case multiple processes send a request to *Server*. As the process structure of software may be different, it is required that a runtime enforcement framework can automatically generate the enforcement modules for various process structures, and can keep track of causality. This decreases the programmers' effort to utilize a runtime enforcement framework for software systems with various process structures.

4. **Language transparency in specifications**: As for the process structure, we claim that a runtime enforcement framework must offer specification languages

that are transparent from the implementation languages of the base software. This increases the reusability of specifications, and decreases programmers' effort in maintaining the specifications, if the implementation languages of the base software change.

5. **Automatic generation of runtime enforcement components for various implementation languages**: A runtime enforcement framework must offer a compiler that can generate and integrate components in a variety of languages.

6.1.3 Shortcomings of the Existing Runtime Enforcement Frameworks

Although there are numerous runtime enforcement frameworks in the literature, they are usually dedicated to support a specified process structure and/or implementation language. In the following, a brief evaluation of some of the existing runtime enforcement frameworks are provided:

1. Adopting runtime enforcement frameworks [17, 38, 55, 64, 10, 76] that can only support single-language and single-process software. Therefore, they do not fulfill the previously-mentioned requirements. Among these, [38] supports software implemented in C, and the others support software implemented in Java.

   One may aim at employing these frameworks for multiple-process and/or multiple language software. Consider for example the sequence of events depicted in Figure 6.1. For the end-to-end enforcement of events, one could consider dividing this sequence of events into two parts according to the process structure, and verifying them separately.

   This would not, however, provide the desired end-to-end enforcement, since the causal dependency of the events between *Server* and *Client* cannot be checked. Consequently, one cannot be sure that the occurrence of the event, say *store* on object *aStorage* in Figure 6.1, is a result of the occurrence of the event *save* in *Client*, especially if there are multiple client processes sending the store request to the process *Server*.

   To overcome the above problem, one should develop a dedicated runtime enforcement framework for checking the causal dependency of event sequences occurring in individual processes. However, since the inter-process communication can be arbitrarily complex, this would largely increase the verification

effort. Moreover, the solution may be too specific and therefore can be difficult to reuse it in different software architectures.

Instead of seeking for dedicated solutions, it is considered more feasible to develop a generic runtime enforcement framework that can deal with any process structure in distributed occurrence of events.

2. Adopting runtime enforcement frameworks [60, 48, 97] that are tailored to inter-process communication at the middleware level.

   If end-to-end enforcement of distributed software is aimed at, additional runtime enforcement frameworks must be utilized that can deal with verifying events within a subsystem at the application level. Even so, one needs to implement dedicated algorithms to check the causality of events that are distributed across multiple processes. Therefore, the requirement end-to-end enforcement is not satisfied by these frameworks. Moreover, because the enforcement is limited to inter-process communications, specifications are distribution-sensitive. Finally, the automatic generation of enforcement components for arbitrary process structures is not fulfilled by these frameworks.

3. Adopting runtime enforcement frameworks [71, 86] that support multiple-process software.

   These frameworks can specify events executing within and across different processes, but without fulfilling the requirement distribution-transparency in specifications. Therefore, if the distribution of software changes, programmers are forced to adapt the specifications accordingly; and this may be an error-prone and time-consuming task. Moreover, these frameworks are also limited to software developed in the Java language.

6.2 Supporting Multiple-Process Java Software in EventReactor

As it is shown in the previous chapters, the EventReactor language is independent of the programming languages, and its compiler supports software implemented in the Java, C and .Net languages.

If the base software is implemented in Java and makes use of Java-RMI [46] as the middleware, the EventReactor language facilitates defining distribution-transparent specifications for it. To infer whether the base software is distributed or not, the compiler of the EventReactor language analyzes it and accordingly generates code.

Assume, for example, that our illustrative example is Java-RMI based software. Since EventReactor supports distribution- and language-transparent specifications, the specifications in Listings 5.1, 5.2, 5.3 and 5.4 can be reused for out illustrative example.

The following subsections explain how the compiler and the runtime behavior of the EventReactor language support distributed Java software. Along this line, a discussion about the runtime overhead and the limitations of the EventReactor language is presented as well.

6.2.1 The Compiler Support for Distribution-Transparent Specifications

The EventReactor compiler must generate code for the runtime environment and the necessary modules such as notifiers for arbitrary process structure.

If the base software consists of multiple processes, it is possible to execute the runtime environment of the EventReactor language and the notifier modules in separate processes. In this case, the notifier modules make use of inter-process communication to inform the occurrence of events. Naturally, inter-process communication introduces a certain overhead.

To avoid unnecessary overhead, the compiler analyzes the base software to infer whether it is distributed or not, and accordingly generates code. Figure 6.2 provides an overview of the compiler, which supports distribution-transparent specifications.

To avoid repetition, in the following, we only explain the parts of the compiler that differ from the one discussed in Chapter 4.

Input and Output

As shown at the top of the Figure 4.1, the compiler receives the following input: *Event Record(s)*, *Publisher Record(s)*, *Program Code*, *Reactor Type(s)*, *Specification of Reactor Chain(s)*, *Specification of Event Package(s)*, and *Client-Side & Server-side Stubs*.

In RMI-based software, the methods that can be invoked remotely by classes at the client side must be defined in interfaces that extend the interface *java.rmi.Remote*. These interfaces must also be implemented by the classes at the server side. The Java-RMI compiler generates server-side and client-side stubs accordingly, which are shown in the figure as *Client-Side & Server-Side Stubs*.

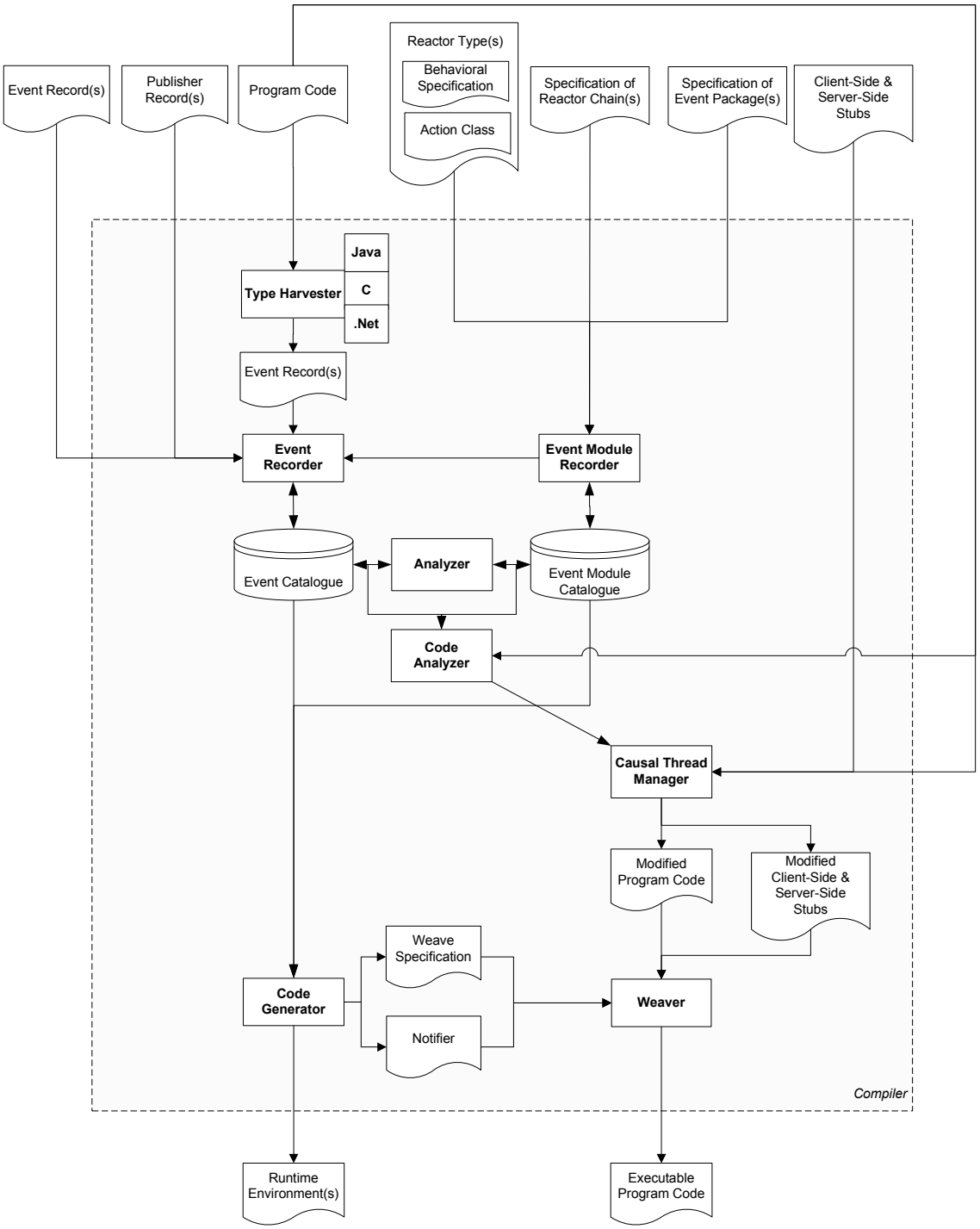

Figure 6.2: An overall view of the EventReactor compiler with the support for distribution-transparent specifications

The compiler modifies *Client-Side & Server-Side Stubs* to keep track of the causality of events among clients and servers.

As its output, the compiler generates *Executable Program Code* and *Runtime Environment(s)*.

Code Analyzer

From *Event Catalogue*, *Code Analyzer* retrieves the event records that represent the predefined events. From *Event Module Catalogue*, *Code Analyzer* retrieves the event modules that refer to the actual event. To infer whether the events that form the input interface of an event module are distributed across multiple processes, *Code Analyzer* performs a two-phase analysis:

1. **Program Analysis**: A Java program has a method named *main*, which starts the execution of the program. If the base software is distributed, to run client and server subsystems, there will be multiple *main* methods. *Code Analyzer* retrieves *main* methods defined in *Program Code* and for each retrieved method, it computes its call graph [83].

   A call graph is a directed graph, which represents a call-relationship between two methods. The module *Code Analyzer* makes use of the Soot framework to construct call graphs. The details can be found in [87].

   In constructing a call graph, the Soot framework does not distinguish between the objects that are instantiated at the client and server sides. Therefore, a call graph may also contain the methods that are executed at the server-side.

   Code Analyzer traverses the constructed call graph in a depth-first search [23] manner to conclude two facts: (a) the specified event occurs in the control flow of the method *main*. In this case, the method main is tagged as relevant. (b) A remote method (i.e. a method defined in a remote interface) is invoked in the control flow of the method *main*. This indicates that the main thread of execution may span across multiple processes at runtime. In this case, the method *main* is tagged as the client-side of distribution, and the remote methods are identified as the points where the thread of execution may span across multiple processes.

2. **Specification Analysis**: for each specified event module, *Code Analyzer* checks if the events that form the input interface of the event module correspond to the previously identified remote methods. If so, the event module is tagged as distributed.

If not, the call graph of the identified remote methods is constructed and the depth-first search is performed to infer if any of the specified events occurs during the execution the remote method. If so, the event module is tagged as distributed, because the event that it groups occur in a thread of execution which spans across multiple processes.

All other event modules that refer to the events published by the tagged event module are also tagged as distributed.

The remote methods which do not refer to any event of interest are ignored.

Code Analyzer stores information about the tagged event modules in *Event Module Catalogue*.

Causal Thread Manager

For each tagged *main* method, the module *Causal Thread Manager* assigns a unique identifier to the physical thread executing the method *main*. This is used as the identifier of the causal thread. In the current implementation, the unique identifier of the causal thread is stored at the Java class ThreadLocal[44] as a local variable.

Program Code is also changed such that when the execution of the method *main* terminates, the systemic event indicating the termination of the thread is sent to the runtime environment of the EventReactor language.

To maintain the unique identifier of causal threads across processes, *Causal Thread Manager* takes the same approach as [93]. The implementation of *Causal Thread Manager* supports older as well as the newest version of Java-RMI in which the functionality of server-side stub is inserted within the server class. In this case, for each remote method identified by *Code Analyzer*, the module *Causal Thread Manager* adds a new method that wraps the remote method. The wrapper has the same signature as the wrapped method but with one additional parameter holding the causal thread identifier received from the client-side stub. Pseudo code of the wrapper is shown in Listing 6.1.

The wrapper code first binds the physical thread of the server-side subsystem to the causal thread, and thereafter invokes the wrapped method. After the termination of the execution of the wrapped method and before the return of the call to the client thread, the wrapper code unbinds the server thread from the causal thread. For the older versions of Java-RMI the same modifications are applied to skeleton modules, only the wrapper method wraps the stub method corresponding to the remote method.

```
1 remoteProc (long causalThreadID,...) {
2   calleePhysicalThread.causalThreadID := causalThreadID
3   invoke remoteProc (...)
4   calleePhysicalThread.causalThreadID := null
5 }
```

Listing 6.1: Spanning the causal thread of execution to server-side

For each tagged client-side method *main* and for each identified remote method, *Causal Thread Manager* changes the client-side stubs so that the stub methods invoke the wrapper method in the server-side stub with the causal thread identifier that has already been assigned to the physical client-side thread. Listing 6.2 shows the pseudo code for a modified client-side method.

```
1 remoteProc (...) {
2    invoke remoteProc (callerPhysicalThread.causalThreadID ,...)
3 }
```

Listing 6.2: Spanning the causal thread of execution from client-side

The module *Weaver* of the compiler accepts the modified *Program Code* and the modified *Client-Side & Server-Side Stubs* as input, and generates an integrated code as explained in Chapter 4.

Code Generator

If any of the specified event modules is tagged as distributed, *Code Generator* creates two sorts of runtime environment modules for the EventReactor language: (1) The one that is executed in a dedicated process called *Enforcement Server*, and (2) The ones that are executed locally in application processes.

The event modules which are not tagged as distributed are managed by local runtime environments, which also communicate with the server runtime environment to inform it of the occurrence of events.

6.2.2 Runtime Behavior

The runtime behavior explained in Section 5.3 is also valid for the distributed case. There are some differences between the single process and multiple process implementations of the runtime environment due to inter-process communication. In the

following, we provide a generic explanation of the runtime behavior for the multiple process case.

If there is any event module tagged as distributed, a process called *Enforcement Server* is created, which executes the runtime environment of the EventReactor language.

To manage the instances of the event modules that are not distributed, and to communicate with the runtime environment executing in *Enforcement Server*, there is an instance of runtime environment locally executing in each application process. These make use of Java-RMI to communicate with their counterpart in *Enforcement Server*. Therefore, the communications are synchronous. We assume that Java-RMI is a reliable middleware and the problems such as event loss and out-of-order events are negligible.

Assume that the event e is detected in the thread t executing within an application process. The thread t can be a local thread in the process, or a causal thread.

Notifier, which is bound to the corresponding publisher object p, creates an instance of the data structure *RTEvent* and initializes the necessary static and dynamic contextual information. This includes p and t, which are the unique identifier of the publisher and the unique identifier of the corresponding thread, respectively.

Notifier announces the event by sending the instance of *RTEvent* to *Runtime Environment* that is executing in the application process. Using the protocol explained in Chapter 5, *Runtime Environment* accesses *Event Catalogue* to check if the published event is known in the system. If the evaluation results in TRUE, it retrieves the corresponding event modules from *Event Module Catalogue*. There may be multiple corresponding event modules. Some of these may be tagged as distributed.

If none of the event modules are distributed, the event is processed as it is explained in Chapter 5. Otherwise, *Runtime Environment* that is locally executing in the application process, determines the execution order of the event modules by referring to the constraint *precede*, if there is any[2]. If an event module is tagged as distributed, the local *Runtime Environment* forwards the instance of *RTEvent* to its counterpart executing in the process *Enforcement Server*.

When the event is processed, the instance of *RTEvent* is returned to the local *Runtime Environment*. This facilitates accessing the return context of the event in the application process.

[2]The current version of EventReactor does not support the constraints *ignore* and *override* for distributed event modules.

6.2.3 An Illustration of Runtime Behavior

This section makes use of the document-editing software to illustrate the runtime behavior of the EventReactor language for multiple process software.

Assume that the document-editing software is compiled against the specifications defined in Listings 5.1, 5.2, 5.3 and 5.4. We also assume that the compiler of EventReactor tags the event module *document_eventmodule* as distributed.

Figure 6.3 illustrates the runtime view of the document-editing software. Here, the emphasis is on the application objects which are the publishers of the events. In the figure, the runtime environments are distinguished with suffixes *(C)*, *(S)* and *(E)*, standing for *Client*, *Server* and *Enforcement*

Consider the following scenarios:

A publisher is created

The objects *aDocument* and *aStorage* are the publishers of the events of interest. Upon their creations, an instance of *Notifier* is created and bound to them.

A sequence of causally-dependent events occurs

Assume, for example that in the main thread of execution within *Client*, the method *save* is invoked on *aDocument*. This results in the detection of the event *save*.

Notifier(D) creates an instance of RTEvent and initializes its fields. The unique identifier of *aDocument* is specified as the unique identifier of the publisher of the event. During the compilation process, the main thread of execution in *Client* is detected as a causal thread that spans across multiple processes. Therefore, a unique identifier is assigned to it. This value is kept in the corresponding field of the instance of *RTEvent*.

Runtime Environment(C) identifies that the event module *document_eventmodule* is registered for the event. Since this event module is tagged as distributed, *Runtime Environment(C)* forwards the instance of *RTEvent* to its counterpart *Runtime Environment(E)* in the process *Enforcement Server*. Further, the same procedure as discussed in Section 5.3 is used in creating the event module table and initializing it.

To process events that are published by the event module *document_eventmodule*, an instance of *recovery_actions_eventmodule* is created by *Runtime Environment(E)*.

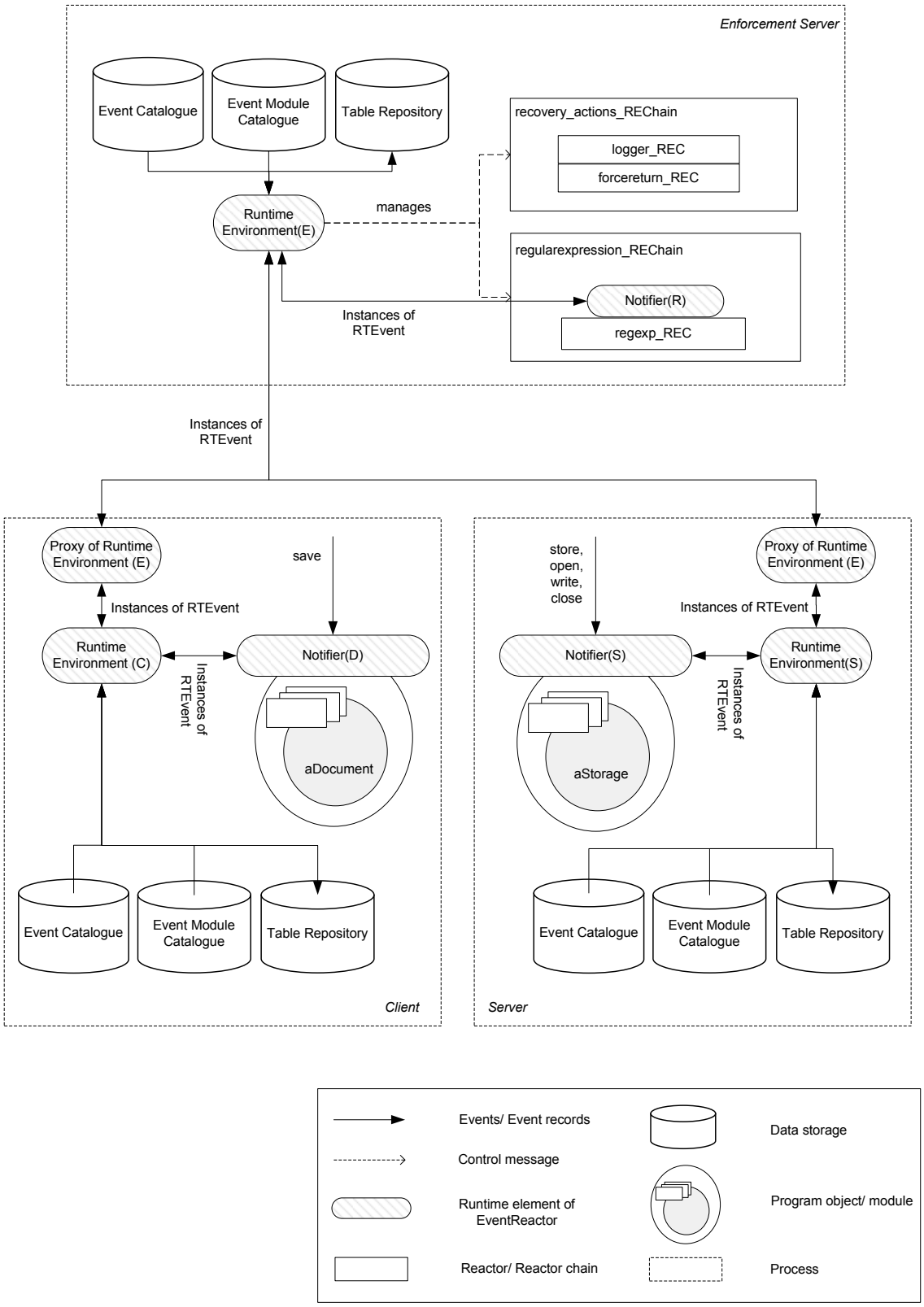

Figure 6.3: A runtime view for the distributed document-editing software

When the event *save* is processed in *Enforcement Server*, the instance of *RTEvent* is returned to *Runtime Environment(C)*, which provides it to other event modules, if any.

Assume that as the result of the execution of the method *save*, the method *store* is invoked on the object *aStorage*. *Notifier(S)* creates an instance of *RTEvent* to represent the event, and initializes its fields. The unique identifier of the current causal thread is assigned to the corresponding field in *RTEvent*. This has the same value as the one assigned to the event *save* in the process *Client*, because the causal thread spanned across these two processes.

The event *store* is forwarded to *Runtime Environment(E)*, and is processed.

An event of interest occurs in a local thread of execution

Now, assume for example that the event *store* occurs in a local thread within the process *Server*.

An instance of *RTEvent* is created by *Notifier(S)*. This time the unique identifier of the local thread of execution is assigned to the event. The event module *document_eventmodule* has registered for the event *store* and it is tagged as distributed. Therefore, the event is forwarded to *Runtime Environment(E)*.

Listing 5.1 specifies the event module as *perinstancethread*. Therefore, *Runtime Environment(E)* inserts a new row in the event module table, with the unique identifier of the local thread in its column *col_thread*. A new instance of the reactor chain *regularexpression_REChain* is created to process the event.

There are multiple clients and/or servers

In distributed software, there can be multiple clients that make use of the services provided by a server. Individual clients initiate their own causal thread of execution with different identifiers. This helps to distinguish between the events published by different clients.

The number of servers depends on the application program. Assume, for example, the document-editing software evolves such that documents are stored in multiple servers. Each server has an instance of class *Storage*.

Since Java-RMI is used as the underlying middleware, clients communicate with servers in a synchronous manner. This means that a request to store a document

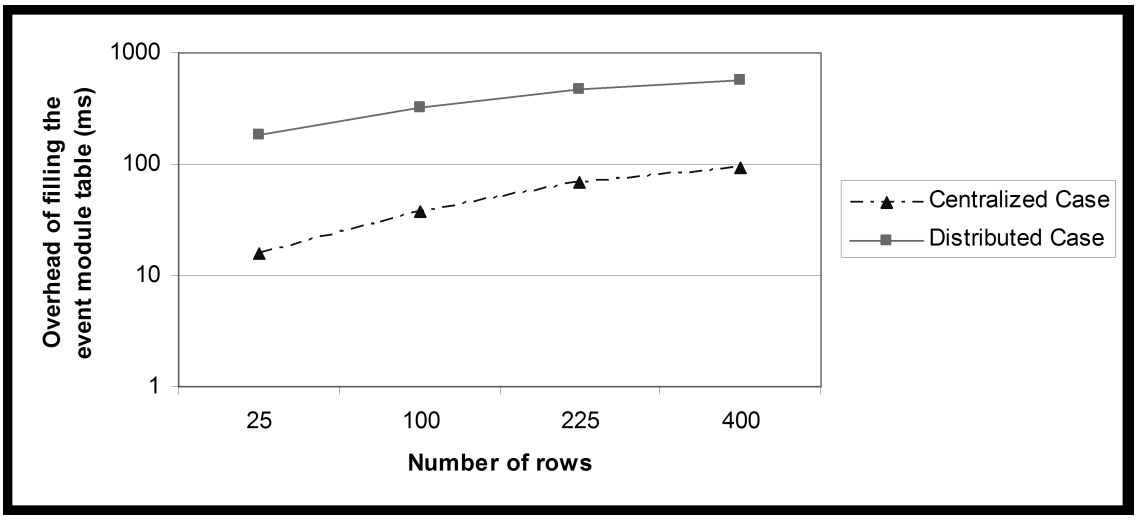

Figure 6.4: The runtime overhead for filling event module tables

within two servers is handled by one causal thread of execution, which sequence spans over two servers. The processing is then carried out sequentially.

Since there are two instances of class *Storage*, with each of them has its own unique identifier, two separate rows are created in the event module table. A separate instance of the reactor chain *regularexpression_REChain* is created for each row. The event module table is maintained in the process *Enforcement Server*.

6.2.4 Evaluation

We measured the runtime overhead of EventReactor in verifying the specified sequence of events for a distributed implementation, as well as a centralized implementation of the illustrative example.[3]

The runtime overhead is influenced by the number of publishers that are participating in a thread of execution. We consider four cases: 5, 10, 15 and 20 instances per class *Document* and *Storage*. Since the specified regular expression is verified for all the possible participation of these objects in a single thread, there are 25, 100, 225 and 400 combinations for which the verification must be carried out.

The runtime overhead is divided into two parts: time to create and fill the event module tables, and time to process events for each row of the event module tables.

[3]The evaluation is performed using a preliminary implementation of the EventReactor language.

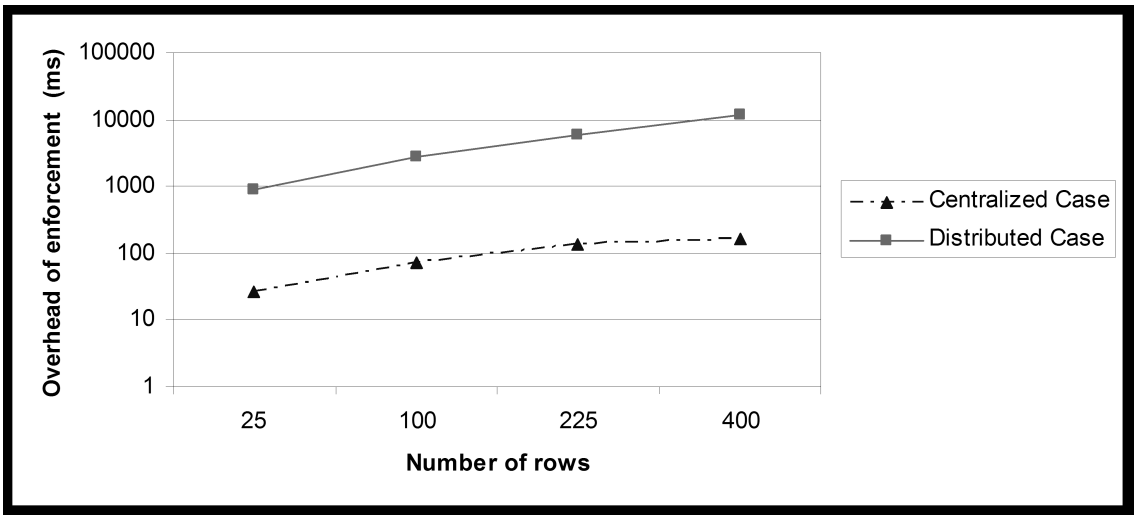

Figure 6.5: The runtime overhead for processing events

The evaluation was conducted on a 2.00 GHz Intel Core 2, with 2GB RAM running the Hotspot Client JVM version 1.6.0 under Windows XP. All on a single machine.

Figures 6.4 and 6.5 show the results of the evaluation. The X-axis represents the number of rows in the table created for the event module *document_event module*. The Y-axis represents the time for filling the tables and/or for verifying the execution in milliseconds. The graph shows the overhead in a logarithmic scale.

Maintaining the table in a separate process for a distributed software, imposes an extra overhead due to the inter-process communication. There are different optimization techniques that can be employed. For example, caching techniques may be used to keep a copy of the tables in each application process, so the amount of inter-process communication decreases. However, in general there is no single optimum solution for all possible variations.

6.2.5 Limitations in Supporting Distribution Transparency

We identify two major shortcomings in the proposed solution for supporting distribution-transparent specifications:

1. In RMI-based software, a remote interface may be bound with a local or a remote server object, dynamically. *Code Analyzer* of EventReactor does not distinguish between local and remote objects. Therefore, sequences of events will be always considered distributed. This does not influence the functionality,

but increases the runtime overhead due to maintaining event module tables in separate processes. We identify two possible solutions to this problem:

First, there are techniques known as points-to analysis [57] that statically determine the set of objects pointed by a reference variable or a reference object field. The points-to analysis technique can be applied for distinguishing remote and local objects from each other. Second, it is possible to give up the distribution-transparency of specifications in a controlled manner, by including information about the distribution of modules.

2. To support distributed multiple-language software, code analyzer must extract the call-graph of software that expands across modules developed in various languages. However, it may not always be possible to attain an adequate call graph. For example, assume that there is a C function that makes use of Java-JNI API to invoke a Java method whose name is defined as a variable. If the value of the variable is determined at runtime, it is not possible to extract an adequate call graph before the actual execution of software at runtime. The same problem exists in single-language software that makes use of a reflection technique [31] to invoke a method.

6.3 Supporting Distribution-Sensitive Specifications in EventReactor

To overcome the above mentioned limitations, it is possible to relax the distribution-transparency requirement by enabling the programmers to specify the distribution information.

Listing 6.3 shows a distribution-sensitive specification for the document-editing software. As line 37 shows, programmers can make use of the keyword `distributed` to specify that the events forming the input interface of an event module are distributed. The distribution-transparency can nevertheless be preserved for the specifications of reactor chains.

The specifications are input to the EventReactor compiler. Since no code analysis is performed by the compiler, it is also not possible to automatically maintain the causal thread of execution among multiple processes. We consider two solutions for this matter.

First, programmers may manually modify the program code to keep track of the causality of events, by putting the unique identifier of the causal thread in a memory location that is known by the runtime environment of EventReactor. Second,

programmers may wrap the inter-process communication code in a client- and server-side stub modules as it is the case in Java-RMI. In this case, the module *Causal Thread Manager* in the EventReactor compiler can be extended to modify these stub modules for passing the unique identifier of causal threads across the processes.

```
1  eventpackage user_request{
2      selectors
3          save_event = {E |
4                              isBeforeExecution(E, M),
5                              isMethodWithName(M,'save'),
6                              isClassWithName(C,'Document'),
7                              isDefinedIn(M, C)};
8
9          store_event = {E |
10                             isBeforeExecution(E, M),
11                             isMethodWithName(M,'store'),
12                             isClassWithName(C,'Storage'),
13                             isDefinedIn(M, C)};
14
15         open_event = {E |
16                             isBeforeExecution(E, M),
17                             isMethodWithName(M,'open'),
18                             isClassWithName(C,'Storage'),
19                             isDefinedIn(M, C)};
20
21         close_event = {E |
22                             isBeforeExecution(E, M),
23                             isMethodWithName(M,'close'),
24                             isClassWithName(C,'Storage'),
25                             isDefinedIn(M, C)};
26
27         write_event = {E |
28                             isBeforeExecution(E, M),
29                             isMethodWithName(M,'write'),
30                             isClassWithName(C,'Storage'),
31                             isDefinedIn(M, C)};
32
33     eventmodules
34         document_eventmodule := {save_event, store_event, open_event,
35                                    write_event, close_event}
36                                  <-
37                                  distributed perinstancethread
38                                  {regularexpression_REChain('
39                                (save_event store_event open_event write_event+ close_event)*'
40                                  )} ;
41 }
```

Listing 6.3: A distribution-sensitive specification of event modules

6.4 Conclusion and Future Work

This chapter defines requirements that should be fulfilled by a runtime enforcement framework to facilitate runtime enforcement of multiple-language and/or multiple-process software. These are: a) distribution transparency in specifications; b) end-to-end enforcement; c) automatic generation of enforcement modules for arbitrary process structures; d) language transparency in specifications; and e) automatic generation of runtime enforcement components for various implementation languages. The chapter argues that fulfilling these requirements helps to increase the reusability of specifications for distributed and/or multiple language software.

The chapter discusses the degree to which these requirements are addressed by the current implementation of the EventReactor language. First a solution is proposed that fulfills the first four requirements. Here, the compiler of the EventReactor language only supports the Java software that makes use of Java-RMI as middleware.

The chapter discusses that it may not be feasible in practice to fulfill all the mentioned requirements for distribute and/or multiple-language software that makes use of various middleware and/or inter-process communication techniques. As a solution, the chapter proposes to relax the distribution-transparent specifications in a controlled manner.

In the current implementation of the EventReactor language, if an event module is specified/tagged as distributed, its corresponding table is maintained in a shared process called *Enforcement Server*. One may argue that maintaining this information in a separate process results in the well-known problem of shared memories (e.g. bottleneck). As it is discussed in Section 5.2.2, there are also other alternatives for this matter, which may impose more runtime overhead due to the amount of the interprocess communications that they require. The degree to which selected events are distributed may influence the imposed runtime overhead. The evaluation of these alternatives is considered as future work.

Currently, the module *Causal Thread Manager* provided by the EventReactor compiler only considers the physical thread of execution that is created by Java virtual machine for executing the method *main*. However, there might be other threads of execution in multi-threaded software, which are created and destroyed explicitly by software. As a future direction of the implementation of the EventReactor language, it is considered to extend *Causal Thread Manager* to recognize such threads of execution so that a casual thread can be maintained for each of them.

The EventReactor language provides three constraints *precede*, *override* and *ignore* for event modules. The constraints *precede* and *override* are evaluated statically by the compiler and are not influenced by the distribution of software. The oper-

ator *ignore* is evaluated at runtime based on the stack of executing events. In the distributed case, a global stack must be maintained across multiple processes for each causal thread of execution. This is also regarded as a future direction in the implementation of the EventReactor language.

Both proposed solutions for the runtime enforcement of distributed software are limited to the processes that communicate in a synchronous manner. However, there is a vast number of distributed software that makes use of an asynchronous communication technique. Since in such software events may occur concurrently, a suitable formalism must be adopted to express the desired properties of software in the specifications. The algorithm to analyze the code remains the same for both synchronous and asynchronous cases. An existing algorithm such as the one presented in [65, 30] may be employed to keep track of causality of events. The runtime environment must deal with the well-known problems of asynchronous communications such as global clocks, or out of order events. Supporting these features is considered a future direction of the implementation of the EventReactor language.

We consider extending the offered linguistic constructs of the EventReactor language with distribution-sensitive specifications, such as process and the machine identities. Such constructs enable programmers to have more control over the distribution of the runtime enforcement components.

Chapter 7

A Case Study for the Evaluation of the EventReactor Language

As it is discussed in Chapter 2, since the specification languages of runtime enforcement frameworks adopt the elements of their underlying languages, they may fall short in representing various runtime enforcement concepts naturally. To overcome the limitations, the computation model Event Composition Model and its implementation the EventReactor language have been introduced.

This chapter presents an evaluation of the EventReactor language from the perspective of its support in natural representation of runtime enforcement concepts. For this matter, the chapter makes use of an example runtime enforcement technique named Recoverable Process, which aims at providing fault-tolerant processes. This technique was introduced for the TRADER project [92].

The chapter discusses some of the possible implementations of Recoverable Process in existing runtime enforcement frameworks and programming languages. A detailed discussion about the shortcomings of such implementations is provided, followed by a possible implementation in the EventReactor language, which overcomes the shortcomings.

Section 7.1 explains the Recoverable Process technique and its application to a media player software. Section 7.2 discusses an implementation of Recoverable Process in the C language, which was provided for the TRADER project.

Section 7.3 outlines a possible implementation of Recoverable Process in an existing runtime enforcement framework, and discusses the shortcomings of this implementation in providing a natural representation for the concepts of Recoverable Process.

Section 7.4 discusses a possible implementation of Recoverable Process in an existing aspect-oriented language, and discusses the shortcomings of this implementation in providing a natural representation for the concepts of Recoverable Process.

Section 7.5 outlines a possible implementation of Recoverable Process in the EventReactor language. It explains the extendability of the language with user-defined events, which plays an important role to represent the concepts of Recoverable Process naturally.

Finally, Section 7.6 outlines the conclusions.

7.1 An Illustrative Runtime Enforcement Technique

This section first provides an example runtime enforcement technique called Recoverable Process, which detects the failures of processes and recovers them. Secondly, it applies this technique to the development of a media player software.

7.1.1 Recoverable Process

In [88], a runtime enforcement technique called Recoverable Process is published to make processes fault-tolerant. Here, processes are monitored to detect their failures, and a failed process is restarted along with other processes that are semantically related to it.

The technique treats various failures of a process, for example, the crash of a process, and a deadlock within a process. If processes exchange messages among each other, a failed process may not be available to receive the messages from other processes. Therefore, Recoverable Process also considers the recovery of such messages through message queuing and dispatching.

Recoverable Process assumes that after recovery, processes can continue their normal operation. This implies that the failures must have a transient nature, i.e. the chance that a failure reoccurs after a restart is low.

Figure 7.1 provides an overall view of the concepts in Recoverable Process. For the sake of brevity, [88] assumes that only child processes that have a common parent process can be recovered; this thesis also makes the same assumption.

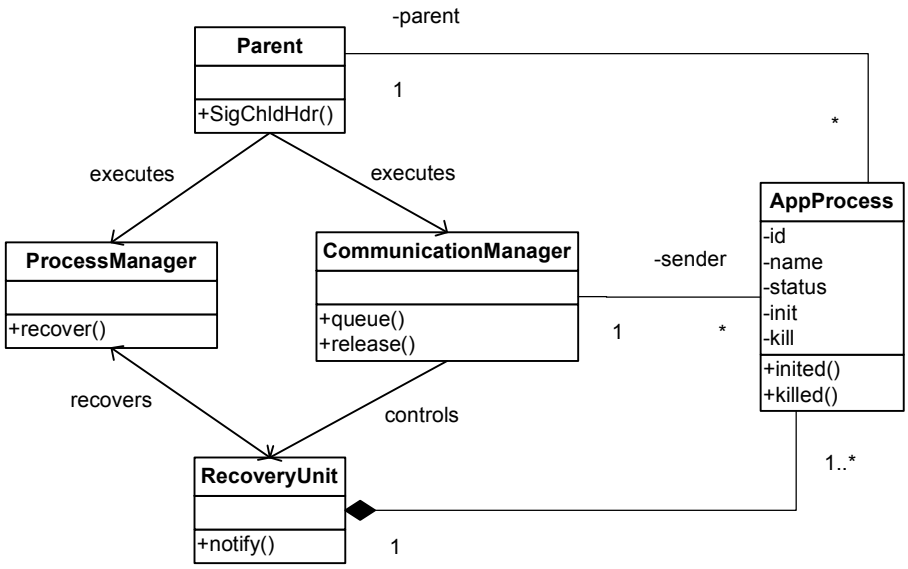

Figure 7.1: The concepts of the Recoverable Process technique

AppProcess represents a child process, and has the attributes *id*, *name*, *status*, *init* and *kill*. The attribute *id* is the unique identifier of the process, which is generated by the operating system. The attribute *name* is the developer-specified name of the process. The attribute *status* is the execution state of the process, which can either be *running*, *terminated* or *under-recovery*. The attributes *init* and *kill* are the methods that create or kill the process, respectively.

The events *initiated* and *killed*, which are shown as operations in the figure, occur if the process is created or killed, respectively.

The concept *RecoveryUnit* of Recoverable Process represents a group of child processes that must be recovered together. *RecoveryUnit* also detects failures in the grouped processes and triggers an event when a failure is detected.

The concept *Parent* represents the parent process of *AppProcess*. When a child process is killed, the method *SigChldHdr* of *Parent* triggers the event *killed* in the corresponding instance of *AppProcess*.

The concept *Parent* executes the module *ProcessManager*, which performs a recovery action if *RecoveryUnit* triggers a process-failure event. *ProcessManager* changes the status of processes that form the recovery unit to *under-recovery*, and restarts them. After a successful recovery, it changes the status of the processes to *running*.

If other processes send messages to a process that is being recovered, *CommunicationManager* queues the messages and sends them to the process after it is successfully recovered. *CommunicationManager* is also executed by *Parent*.

For the sake of brevity, this thesis only considers the (unexpected) termination of processes as failure, and does not discuss the details to distinguish between the normal termination of a process by the user and the un-expected termination of a process. In addition, we do not discuss the implementation of *CommunicationManager*.

7.1.2 An Application of Recoverable Process

In this section we discuss an application of the Recoverable Process technique to an example media-player software[1]. An abstract block diagram of the media player software is shown in Figure 7.2.

The software is structured around five processes. The surrounding rectangle is the parent process *Runner*, which is initiated by the module *Main*. The parent process creates the child processes and is notified if the child processes are destroyed. The four contained dashed rectangles represent the child processes.

The process *UserInterface* runs the module *GUI* that provides a user interface for the media player software. The process *MPCore* runs the module *Core* that implements the main functionality of the media player software, e.g. reading a media file and buffering it, separating audio and video channels, synchronizing audio and video channels, and handling input commands. The processes *Audio* and *Video* respectively run the modules *Libao* and *Libvo* to play audio and video channels. The arrows in the figure represent the messages that are exchanged among processes.

There are various ways to apply Recoverable Process to the media player software. For example as Figure 7.2 shows, the processes *Audio*, *Video* and *UserInterface* receive information from the process *MPCore*. If *MPCore* is killed, the other child processes cannot continue their operation as well. Therefore, one may group all the child processes as one recovery unit such that if *MPCore* is killed, the recovery unit reports a failure. This is known as **global recovery** [88]. One may also assume that if any of the processes *Audio*, *Video* and *UserInterface* is killed, it can be restarted individually. In this case, it is possible to define separate recovery units for the individual recovery of *Audio*, *Video* and *UserInterface*. This is known as **local recovery** [88] which aims at improving the availability of non-faulty processes.

[1]This is a reduced version of the MPlayer software [70] that was used in the TRADER project as the case study.

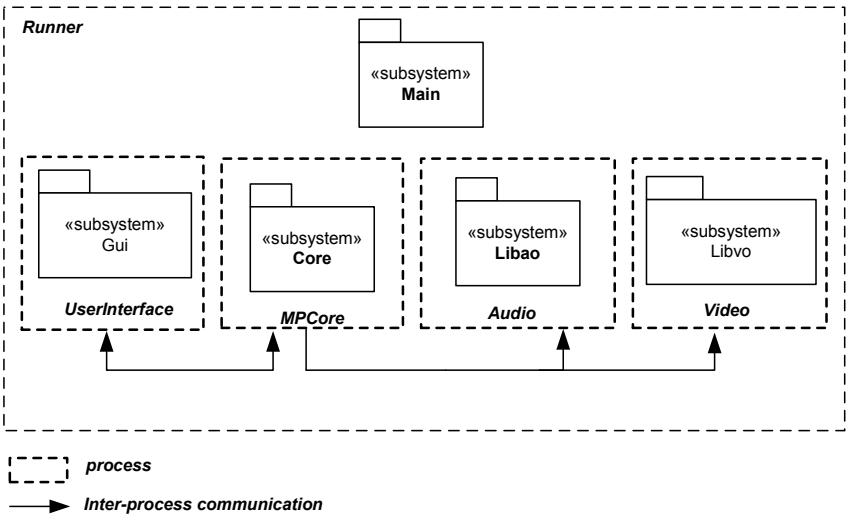

Figure 7.2: An abstract block diagram of the media player software

7.2 Implementing Recoverable Process in Imperative Languages

Within the TRADER project, the Recoverable Process technique for the MPlayer software is implemented in the C language [88].

In this implementation, the modules *ProcessManager* and *CommunicationManager* are defined as separate modules. These modules also maintain information about the processes that must be recovered, and the recovery units to which they belong. The code to detect the failure of the child processes, and the invocations to the modules *ProcessManager* and *CommunicationManager* to activate the recovery actions upon the failure of the child processes, are scattered in the base software .

This implementation of Recoverable Process falls short to represent the concepts of Recoverable Process naturally because of the following reasons. First, there is no explicit and modular representation of processes of interest and recovery units; their definition is tangled in the modules *ProcessManager* and *CommunicationManager*. This also reduces the modularity and compose-ability of the modules *ProcessManager* and *CommunicationManager*, because they cannot be reused for different processes and recovery units.

Second, the concepts of Recoverable Process are not modularized from the base software, because programmers must write code to invoke the functionality of *ProcessManager* and *CommunicationManager* from within the base software.

Third, the concepts are considered at a too low level of an abstraction, since they are expressed in the C language. This reduces the comprehensibility of the represented concepts and their reusability for software developed in other languages.

7.3 Implementing Recoverable Process in Existing Runtime Enforcement Frameworks

To represent the concepts of Recoverable Process such that the representations are modular, compose-able and are at right abstraction level, we try to make use of an existing runtime enforcement framework.

To implement the Recoverable Process technique in existing runtime enforcement frameworks, the specification languages must facilitate natural representation of the following concepts: a) the child processes of interest, b) *RecoveryUnit* as an abstraction over a group of related child processes, and c) the recovery action provided by *ProcessManager*. The specification languages must also facilitate the composition of these concepts so that the Recoverable Process technique is achieved.

There is an increasing number of runtime enforcement frameworks in the literature. For the sake of representation, we make use of JavaMOP to illustrate a possible implementation of the Recoverable Process[2]. The discussions over the shortcomings, nevertheless, can be generalized to other runtime enforcement frameworks.

JavaMOP provides a specification language which makes use of the aspect-oriented language AspectJ as its underlying language. The specification language adopts the pointcut designators of AspectJ to specify the events of interest in the base software. It provides various formalisms to specify the properties of the base software, and is extendable with new formalisms. The specifications of diagnosis and recovery are defined using the elements of the Java language. JavaMOP provides a compiler that translates the specifications to AspectJ code, which can be further integrated with the base software via the AspectJ compiler.

Chapter 2 discusses that JavaMOP, and other evaluated runtime enforcement frameworks, do not facilitate natural representation of the concepts of the Recoverable Process technique. As a result, we must map these concepts to the elements available in the specification language. As it is discussed in Chapter 2 such mappings

[2]Although MPlayer is implemented in the C language, for the purpose of demonstration we make use of a Java-based runtime enforcement framework. Compared to the existing C-based frameworks, the Java-based ones are more mature and they benefit from advanced features of their underlying aspect-oriented language. However, they are still deficient to fulfill our identified requirements for a runtime enforcement system.

suffer from three problems: a) *decreased modularity*, b) *decreased abstraction level*, and c) *decreased compose-ability*. To illustrate these problems, the following subsections provides specifications of the concepts of Recoverable Process in the JavaMOP language.

7.3.1 Decreased Modularity

The representation of a concept can be scattered across the multiple underlying language elements due to not directly representing the concepts of interest. Moreover, the representation of a concept may also get tangled with the representation of other concepts.

Assume for example that we would like to utilize Recoverable Process for the global recovery of the media player software. Listing 7.1 shows an excerpt of JavaMOP specification for this matter.

As it is defined in Section 7.1.2, in the global recovery we would like to restart the processes *MPCore*, *Audio*, *Video* and *UserInterface* when *MPCore* is killed. Therefore, all these child processes form a recovery unit.

Lines 2 to 16 of Listing 7.1 represent the concept *RecoveryUnit*. Here, the variable `recoveryunit_mapping` maintains the list of child processes that form the recovery unit. The events `e_MPCoreCreated`, `e_AudioCreated`, `e_VideoCreated`, and `e_UserInterfaceCreated` are the events of interest that must be detected in the base software. These represent the creation of child processes of interest. After a child process is created, its unique name and unique identifier is inserted in the variable `recoveryunit_mapping`. This is shown for example in line 5 of Listing 7.1.

The event `e_MPCoreKilled`, specified in line 16, is detected when the process *MPCore* is killed.

Lines 17 to 26 represent the concept *ProcessManager*. In this code block, the child processes *UserInterface*, *Audio* and *Video*, which are also elements of the recovery unit, are killed and the child processes are restarted. We have developed the utility class `RecProcess` for this matter.

Although in Figure 7.1, the concepts *RecoveryUnit* and *ProcessManager* are represented as two modules, we could not preserve their modularity in their specification, and they are tangled with each other in one specification module depicted in Listing 7.1.

This, in the first place, does not match the intentions of the software engineers because the specifications do not preserve the characteristics of the design. Moreover,

the reusability of the concepts decreases, because it is not possible to reuse them independently in other contexts. Last but not least, the comprehensibility of the specifications decreases, because as the number and the complexity of the concepts increase, it may not be easy for the software engineers to comprehend the concepts that are represented within one specification module.

```
1  GlobalRecovery(){
2      HashMap <String, Integer> recoveryunit_mapping = new HashMap<String, Integer>();
3
4      event e_MPCoreCreated after() returning (Integer id): call (Integer Main.createMPCore(..))
5              {this.recoveryunit_mapping.put("MPCore", id);}
6
7      event e_AudioCreated after() returning (Integer id): call (Integer Main.createAudio(..))
8              {this.recoveryunit_mapping.put("Audio", id);}
9
10     event e_VideoCreated after() returning (Integer id): call (Integer Main.createVideo(..))
11             {this.recoveryunit_mapping.put("Video", id);}
12
13     event e_UserInterfaceCreated after() returning (Integer id): call (Integer Main.createUI(..))
14         {this.recoveryunit_mapping.put("UserInterface", id);}
15
16     event e_MPCoreKilled after() : execution (* Main.MPCoreKilled(..)) {
17     Iterator i = this.recoveryunit_mapping.entrySet().iterator();
18                 while(i.hasNext()){
19                     Map.Entry me = (Map.Entry)i.next();
20                     if (me.getKey().equals("MPCore"))
21                         RecProcess.reinitialize("MPCore");
22                     else{
23                         RecProcess.kill(me.getValue());
24                         RecProcess.reinitialize(me.getKey());
25                         }
26                 }
27             }
28 }
```

Listing 7.1: A specification for the global recovery in JavaMOP

7.3.2 Decreased Abstraction Level

If concepts are represented via the elements of the underlying languages, their abstraction level remains very close to the code level. Consequently, the representations become restricted to a specific programming language and platform. This reduces the reusability of concepts for software developed in different programming languages and platforms.

For example, the representations provided in Listing 7.1 make use of the element of Java and AspectJ languages, so they cannot be reused if the base software is developed in another language. This is in contrast with the design of Recoverable Process depicted in Figure 7.1, where no constraints is specified about the language and/or platform of the base software.

7.3.3 Decreased Compose-ability

As Figure 7.1 shows, the Recoverable Process technique contains various concepts that are composed with each other so that the overall process recovery functionality is accomplished. If the specifications adopt the elements of the underlying languages, the composition strategies can be required to be defined and implemented in the underlying languages as well. Similar to the previously defined problems, if the composition language does not support the composition strategies naturally, the complexity of the composition specifications may increase, and the comprehensibility and reuse may decrease. In the following this problem is represented by two examples.

Application-specific semantics in compositions:

Assume for example that we want to apply local recovery to the process *UserInterface*. Listing 7.2 shows a JavaMOP specification to represent the concepts *RecoveryUnit* and *ProcessManager* for this matter.

In line 4, the event `e_UserInterfaceCreated` represents the state when the child process *UserInterface* is created, and accordingly, its name and unique identifier is inserted in the variable `recoveryunit_mapping`.

In line 8, the event `e_UserInterfaceKilled` must be detected in the media player software, when the child process is killed. As a result, in line 9 the process is reinitialized by the method `reinitialize` defined in the application class `RecProcess`.

```
1  UILocalRecovery() {
2      HashMap<String, Integer> recoveryunit_mapping = new HashMap<String, Integer>();
3
4      event e_UserInterfaceCreated after() returning (Integer id):
5                  call (Integer Main.createUserInterface(..))
6              {this.recoveryunit_mapping.put("UserInterface", id);}
7
8      event e_UserInterfaceKilled after() : execution (* Main.UserInterfaceKilled(..))
9              {RecProcess.reinitialize("UserInterface");}
10 }
```

Listing 7.2: A specification for the local recovery of *UserInterface* in JavaMOP

The specifications in Listings 7.1 and 7.2 both specify the process *UserInterface* as an element of their recovery unit. Assume that at runtime the process *MPCore* fails; consequently as it is specified in lines 17 to 26 of Listing 7.1, the processes *Audio*, *Video*, *UserInterface* are killed for the global recovery and the child processes are re-initialized.

the event `e_UserInterfaceKilled`, which is specified in line 4 of Listing 7.2, is detected when the process *UserInterface* is killed for the global recovery. In line 9 the process *UserInterface* is re-initialized. As a result, there will be two processes running as *UserInterface*.

To overcome the above problem, it must be possible to compose the concepts represented in Listings 7.1 and 7.2 with each other, and constrain their applicability to the process *UserInterface*. For this matter, we want for example to specify that if global recovery is being executed on a group of processes, these processes must not be recovered locally.

JavaMOP offers a fixed set of operators for the composition of concepts, and it is not possible to tailor the semantics of these operators based on application demands. Consequently, we must provide workaround representations for the application-specific composition operators via the elements that are available in the language.

For example to specify the above-mentioned constraint between global and local recovery, we must merge the Listings 7.1 and 7.2 in one specification module, and make use of Java conditional statements to control the execution of the local recovery. This is shown in Listing 7.3. This workaround, however, reduces the modularity of concepts further and increases their complexity.

```
1  LocalandGlobalRecovery() {
2      ...
3      boolean globalrecovery = false;
4      ...
5      event MPCoreKilled after() : execution (* Main.MPCoreKilled(..))
6              {
7               globalrecovery = true;
8                 //recovery code
9               globalrecovery = false;
10             }
11
12     event UserInterfaceKilled after() : execution (* Main.UserInterfaceKilled(..))
13             {
14              if (globalrecovery == false)
15                  //recovery code
16             }
17     ...
18 }
```

Listing 7.3: A specification for the global and local recoveries

Representing (higher-level) concepts by reusing the existing ones:

Assume that we would like to extend Recoverable Process with a new concept termed *Availability* that reports the time elapsed on the recovery of processes. We would like to represent the concept *Availability* such that it can be applied to the both global and local recovery of the media player software.

Listing 7.4 shows an excerpt of the JavaMOP specification for defining *Availability*. In lines 5 and 8, we define two events `e_started` and `e_finished` that must be mapped to the beginning and the end of the recovery action performed by the concept *ProcessManager* in Listings 7.1 and 7.2. However, because it was not possible to provide a natural representation for the concept *ProcessManager*, it is also not possible to identify the events that occur from within the concept *ProcessManager*. This hinders the compose-ability of the concept *Availability* with *ProcessManager*.

As a workaround, we may represent the concept *ProcessManager* as a class with a method say `recover`, and replace the lines 17 to 26 in Listing 7.1, and line 9 in Listing 7.2 with an invocation on this method. In this case, it is possible to map the events `e_started` and `e_finished` to the states where the execution of this method starts and finishes. Since `recover` is a normal Java method, it can be invoked from within various specifications or from within the base software. Therefore, it is still necessary to specify that only those invocations of `recover` that occur from

within Listing 7.1 and 7.2 are of the interest. However, it is not possible to do so in JavaMOP and other evaluated runtime enforcement frameworks.

```
1  Availability() {
2      long timer = 0;
3      Calendar cal = Calendar.getInstance();
4
5      event e_started before() : //start of the recovery code in Listings 7.1 and 7.2
6        { timer = cal.getTimeInMillis();}
7
8      event e_finished after() : //end of the recovery code in Listings 7.1 and 7.2
9          { System.out.println("The recovery took "+ cal.getTimeInMillis()−timer);}
10 }
```

Listing 7.4: A specification of timing property for the global recovery of processes

7.4 Implementing Recoverable Process in Aspect-Oriented Languages

The shortcomings of the existing runtime enforcement frameworks in representing the concepts naturally may urge us to implement Recoverable Process in an existing programming language.

In this section, we represent the concepts of the Recoverable Process technique in the AspectJ language, which is the underlying language of JavaMOP. The representations are evaluated with respect to their modularity, abstraction level and compose-ability.

7.4.1 Representing the Concept *AppProcess*

Assume that we would like to implement Recoverable Process for the global and local recovery of the media player software. To achieve a modular representation of the concepts, we implement each concept of Recoverable Process as an aspect. The defined aspects are composed with each other to accomplish the desired process recovery.

Listing 7.5 shows an excerpt of the abstract aspect `AppProcessAspect`, which represents the concept *AppProcess*. The aspect defines the attributes `id`, `name`, `init` and `kill`, as it is specified by *AppProcess* depicted in Figure 7.1.

```
1  public abstract aspect AppProcessAspect {
2      public int id;
3      public String name;
4      public Method init;
5      public Method kill;
6
7      abstract pointcut e_Initiated();
8      abstract pointcut e_Killed();
9
10     after(): e_Initiated() { initiated(); ...}
11     after(): e_Killed() { killed(); ...}
12
13     public void initiated() { status="running";}
14     public void killed() { status="terminated";}
15 }
```

Listing 7.5: An aspect representing the concept *AppProcess*

Lines 7 and 8 define the pointcut designators `e_Initiated` and `e_Killed` that represent the process events *initiated* and *killed* depicted in Figure 7.1.

After the event `initiated` occurs, the attribute `status` must be initialized with the value `'running'`. After the event `killed` occurs, the attribute `status` must be initialized with the value `'terminated'`. These are performed by the advice code defined in lines 13 to 14, which are composed with the pointcuts `e_initiated`, and `e_killed`.

Each child process of interest is represented as a sub-class of `AppProcessAspect`. For example, Listing 7.6 shows an excerpt of the aspect `MPCoreProcess` to represent the child process *MPCore* of the media player software. Here, for example, the child process is created by the method `initMPCore` defined in the class `Main`. Line 3 maps the pointcut `e_Initiated` to the invocations on this method. Likewise, the pointcut `e_Killed` is mapped to the invocations on the method `killMPCore` defined in the class `Main`.

```
1  public aspect MPCoreProcess extends AppProcessAspect{
2       ...
3      pointcut e_Initiated(): call (* Main.initMPCore(..));
4      pointcut e_Killed(): call (* Main.killMPCore(..));
5      ...
6  }
```

Listing 7.6: An aspect representing the process *MPCore*

7.4.2 Representing the Concept *RecoveryUnit*

Listing 7.7 defines the abstract aspect `RecoveryUnitAspect`, which represents the concept *RecoveryUnit* of the Recoverable Process technique.

```
1  public abstract aspect RecoveryUnitAspect {
2      public abstract AppProcessAspect[] getProcesses();
3      public abstract AppProcessAspect getInitiator();
4
5      pointcut e_processfailed (AppProcessAspect p) :
6              call(* AppProcessAspect.killed()) &&
7              target(p) && destroyedProcess(AppProcessAspect);
8
9      abstract pointcut destroyedProcess (AppProcessAspect x);
10
11     after (AppProcessAspect process) :
12             e_processfailed(process) {notifyFailure();}
13
14     public void notifyFailure(){}
15 }
```

Listing 7.7: An aspect representing the concept *RecoveryUnit*

In line 2, the method `getProcesses` returns the processes that form a recovery unit.

In line 3, the method `getInitiator` returns the so-called initiator process whose failure must be reported as the failure of the unit. This example assumes that if the initiator process is killed, recovery units must report a failure event.

In lines 5 to 9, the pointcut designators `e_processfailed` and `destroyedProcess` select the points where the initiator process is killed.

The advice code in line 11 triggers a failure-notification event by invoking the method `notifyFailure`.

We can define application-specific recovery units as subclasses of `RecoveryUnitAspect`. Listing 7.8 shows `GlobalRecoveryUnitAspect` that defines a recovery unit for the global recovery of the media player software.

The method `getProcesses` specifies the processes *MPCore*, *UserInterface*, *Audio* and *Video* as the elements of the recovery unit. This is achieved by retrieving the corresponding instance of the aspect `AppProcessAspect` via the operator `aspectOf` of AspectJ.

The method `getInitiator` specifies `MPCoreProcess` as the initiator process.

The pointcut `destroyedProcess` specifies that the destruction of `MPCoreProcess` is considered as the failure of the recovery unit.

Likewise, Listing 7.9 defines a recovery unit for the local recovery of the process *UserInterface*.

```
1  public aspect GlobalRecoveryUnitAspect extends RecoveryUnitAspect{
2      public AppProcessAspect[] getProcesses(){
3          return new AppProcessAspect[] {
4                  MPCoreProcess.aspectOf(),
5                  UserInterfaceProcess.aspectOf(),
6                  AudioProcess.aspectOf(),
7                  VideoProcess.aspectOf()};
8      }
9      public AppProcessAspect getInitiator(){ return MPCoreProcess.aspectOf(); }
10
11     pointcut destroyedProcess(AppProcessAspect x) :
12         target(x) && if(x == MPCoreProcess.aspectOf());
13 }
```

Listing 7.8: An aspect representing the global recovery unit

```
1  public aspect UILocalRecoveryUnitAspect
2                      extends RecoveryUnitAspect{
3
4      public AppProcessAspect[] getProcesses(){ return new AppProcessAspect[]
5                                                 { UserInterfaceProcess.aspectOf() }};
6      public AppProcessAspect getInitiator(){ return UserInterfaceProcess.aspectOf();}
7
8      pointcut destroyedProcess(AppProcessAspect x) :
9        target(x) && if(x == UserInterfaceProcess.aspectOf());
10 }
```

Listing 7.9: An aspect representing the local recovery unit for *UserInterface* process

7.4.3 Representing the Concept *ProcessManager*

Listing 7.10 defines `ProcessManagerAspect`, which represents the concept *ProcessManager* of the Recoverable Process technique. The aspect reacts to the events generated by a recovery unit, and restarts the processes forming the recovery unit.

The pointcut *notified* selects the invocations of the method `notifyFailure` defined within `RecoveryUnitAspect`.

The advice code first re-initializes the initiator process, then retrieves all other processes forming the recovery unit, kills and re-initializes them.

```
1  public aspect ProcessManagerAspect{
2      pointcut notified (RecoveryUnitAspect ru):
3          call(* RecoveryUnitAspect.notifyFailure())&& target(ru) ;
4
5      after(RecoveryUnitAspect ru): notified(ru) {
6          //invoke init method of
7          //the initiator process via reflection
8          ...
9          for (AppProcessAspect p : ru.getProcesses())
10             if (p != ru.getInitiator()){
11               //invoke kill method via reflection
12               //invoke init method via reflection
13             }
14   }
15 }
```

Listing 7.10: An aspect representing the concept *ProcessManager*

`ProcessManagerAspect` refers to `AppProcessAspect` and `RecoveryUnitAspect` instead of their concrete subclasses. Therefore, it can be reused for various child processes and recovery units.

7.4.4 Evaluation

To represent the child processes of interest we need to represent the process-related events *initiated* and *killed*. AspectJ supports a fixed set of events, which are defined in its join point model. New kinds of events must be mapped to the supported events by AspectJ. For example to represent the process-related events *initiated* and *killed*, we assume that there are methods in the base software which create and kill a child process. Accordingly, as lines 3 and 4 in Listing 7.6 shows, pointcuts are expressed to define these events.

The concepts of the Recoverable Process technique are represented via aspects. The inheritance mechanism enables us to provide reusable representations for the concepts of interest. These are defined as abstract aspects. Application-specific representations of the concepts are defined via sub-classes. This is shown for example in Listings 7.5 and 7.6. Such representations are modular.

AspectJ provides a fixed set of operators for the composition of pointcuts and/or advice code. The variety of composition strategies must be mapped to these operators if possible, or must be defined via advice code. This may reduce the modularity of

the composition strategies, because they become scattered across and tangled with aspects. Consider the following example.

As the aspects `GlobalRecoveryUnitAspect` and `ProcessManagerAspect` defined in Listings 7.8 and 7.10 show, during the global recovery of the media player software, the process *UserInterface* is killed and re-initialized. `UILocalRecoveryUnitAspect` detects the destruction of *UserInterface* during the global recovery, and notifies it to `ProcessManagerAspect`. Consequently, there will be two processes executing as *UserInterface*, because a new process is also created by `ProcessManagerAspect`.

To prevent the above situation, we must compose `GlobalRecoveryUnitAspect` and `UILocalRecoveryUnitAspect` such that `UILocalRecoveryUnitAspect` does not publish a failure event if *UserInterface* is killed during the global recovery. This constraint can be represented as the clause "`!cflow(adviceexecution() && within(RecoveryUnit))`", which must be conjuncted to the pointcut `destroyedProcess` in Listing 7.9. The tangling of this composition constraint with the aspect `UILocalRecoveryUnitAspect` increases the complexity of the aspects, and makes it fragile to the changes in the composition constraint.

Supporting a fixed set of composition operators may reduce the compose-ability of the concepts further, if the implementation of a composition strategy imposes major changes on the existing aspects. Assume that the local recovery of *UserInterface* must ignore the case that *UserInterface* is killed during the global recovery. However, if the global recovery cannot re-initialize *UserInterface*, the local recovery must try to do so. There are various possibilities to implement this composition constraint.

One possibility is to encode it in the aspect `ProcessManagerAspect`. In this solution also the constraint is tangled with the dentition of other aspects.

Another solution, which is explained below, improves the modularity of composition constraints, but imposes the following two changes:

First, `ProcessManagerAspect` must be replaced with two aspects, so that it is possible to distinguish between the action for the global and local recovery. We name these two aspects as `ProcessManagerAspect4GR` and `ProcessManagerAspect4LR`.

Second, the aspects `GlobalRecoveryUnitAspect` and `UILocalRecoveryUnitAspect` must also be modified so that the method `notifyFailure` is invoked from within them, instead of from their base class.

Listing 7.11 shows an excerpt of the aspect `ProcessManagerAspect4GR`. Here, line 12 initializes a process and assigns the result of initialization to the variable `result`. If the initialization succeeds, the unique identifier of the process, which is generated by the operating system, is returned; otherwise the return value is -1. In lines 13

and 14 if the result equals -1, the method `failedRecovery` is invoked with the name of the failed process as its argument.

Listing 7.12 defines the aspect `CoordinatorAspect` that modularizes the composition constraint. It selects the invocations of the method `failedRecovery` on the instances of `ProcessManagerAspect4GR`. If the failed process is *UserInterface*, it invokes the method `notifyFailure` on the aspect `UILocalRecoveryUnitAspect`. Consequently, the recovery is performed for the process *UserInterface*.

```
1  public aspect ProcessManagerAspect4GR{
2      pointcut notified (GlobalRecoveryUnitAspect ru):
3          call(* GlobalRecoveryUnitAspect.notifyFailure())&& target(ru) ;
4
5      after(GlobalRecoveryUnitAspect ru): notified(ru) {
6              //invoke init method of
7          //the initiator process via reflection
8          ...
9          for (AppProcessAspect p : ru.getProcesses())
10             if (p != ru.getInitiator()){
11                 //invoke kill method via reflection
12                 int result = //invoke init method via reflection
13                 if (result == -1)
14                  failedRecovery(p.name);
15             }
16         }
17     protected void failedRecovery(String name){}
18 }
```

Listing 7.11: An aspect to implement *ProcessManager* for the global recovery

```
1  public aspect CoordinatorAspect{
2      pointcut notified (String name):
3          call(* ProcessManagerAspect4GR.failedRecovery(String))
4          && args(name);
5
6      after(String name): notified(name) {
7          if (name.equals("UserInterface"))
8              UILocalRecoveryUnitAspect.aspectOf().notifyFailure();
9      }
10 }
```

Listing 7.12: An aspect to coordinate the global and the local recoveries

As the above example shows, the implementation of a composition strategy may impose widespread changes on the existing aspects. Applying such changes is time-consuming task for programmers and may be error-prone.

7.5 Implementing Recoverable Process in EventReactor

To represent the concepts of Recoverable Process in the EventReactor language, the following actions must be carried out:

- The process-related events must be defined in the EventReactor language. As Figure 7.1 shows, *initiated* and *killed* are two events of interest that are published for the concept *AppProcess*. In the EventReactor terminology, these are user-defined events, which according to Figure 4.1 are defined via event records.
- The user-defined events must be published to the runtime environment of the EventReactor language. As Figure 7.2 shows, the process *Runner* is the parent process for the processes *MPCore*, *UserInterace*, *Audio*, and *Video*. It creates these child processes, and is informed if they are destroyed. Therefore, we consider *Runner* as the publisher of the events *initiated* and *killed* for the corresponding child processes.
- The publishers of events must be defined in the EventReactor language. As the input *Publisher Record(s)* in Figure 4.1 shows, the necessary information about the publisher of the events must also be provided as a set of Prolog facts.
- Dedicated reactor types must be provided to implement the functionality of the concepts of Recoverable Process.
- Reactor chains and event modules must be defined to represent the concepts of Recoverable Process. According to Figure 7.1, the specified event modules and reactors are also executed by the parent process *Runner*.

The above-mentioned steps are explained in detail in the following subsections.

7.5.1 Declaring Process-Related Events

As the concept *AppProcess* in Figure 7.1 shows, for each child process of interest two events *initiated* and *killed* must be defined. The definitions are carried out via event records. Each event record specifies three attributes to represent the static contextual information of these events. The attribute *name* represents the name of the event. The attribute *process* represents the application-specific name of the

corresponding child process. The attribute *PrologFacts* represents a set of Prolog facts, which are used to select the event.

Listing 7.13 shows an excerpt of the code to define the event *initiated* of the child process *MPCore*. The other events of interest are defined likewise.

```
1 EventRecord event = new EventRecord();
2 event.staticcontext.add("name", "initiated");
3 event.staticcontext.add("process", "MPCore");
4 event.staticcontext.add("PrologFacts", "isEventWithName('e1', 'initiated').
5                                         isProcessWithName('p1', 'MPCore').
6                                         isPublishedBy('e1','p1').");
7
8 EventReactor.define(event);
```

Listing 7.13: Declaring an event

Line 1 defines the variable `event` of type `EventRecord` that is a data type provided by EventReactor. Line 2 specifies `'initiated'` as the name of the event, and line 3 specifies `'MPCore'` as the name of the corresponding process. Lines 4 to 6 specify the Prolog facts. Via the fact `"isEventWithName('e1', 'initiated')."`, we specify `'e1'` as the static unique identifier of the event and `'initiated'` as the name of the event. The character `'.'` is the standard Prolog operator that represents the termination of a fact. Via the fact `"isProcessWithName('p1', 'MPCore')."` we specify `'p1'` as the static unique identifier of the process `MPCore`, and `'MPCore'` as the application-specific named of the process. The fact `"isPublishedBy('e1','p1')."` specifies that the event `'e1'` is published by and the process `'p1'`. The other events of interest must be defined in a similar way.

7.5.2 Publishing the Events

To publish the events *initiated* and *killed* for each child process, we extend the media player software with a class that defines two methods: one for initiating the child process and one for killing it. The media player software is also changed to invoke these methods when needed.

Listing 7.14 shows an excerpt of the class `MPCore`, which defines two methods `initMPCore` and `killMPCore`. In line 4 of the method `initMPCore`, the child process *MPCore* is created and its unique identifier is assigned to the variable `processID`.

To publish the event *initiated*, an instance of the class `RTEvent` is created, and its attributes are initialized with necessary static and dynamic contextual information for the event. For the static contextual information, lines 6 to 11 specify the same

attributes as the one in Listing 7.13, so it is possible to match a published event with a defined event in the language.

For the dynamic contextual information, lines 13 to 15 specify the unique identifier of the corresponding child process, the unique identifier of the publisher process (i.e. the process *Runner*) and the current stack trace of methods whose execution caused the event *initiated* to be published[3].

After the necessary attributes are set, the event is announced to the runtime environment of the EventReactor language.

```
1  public class MPCore{
2      int initMPCore() {
3      ...
4      int processID = \\ create the child Process...
5      ...
6      RTEvent event = new RTEvent();
7      event.staticcontext.add("name", "initiated");
8      event.staticcontext.add("process", "MPCore");
9      event.staticcontext.add("PrologFacts", "isEventWithName('e1', 'initiated').
10                                             isProcessWithName('p1', 'MPCore').
11                                             isPublishedBy('e1','p1').");
12
13     event.dynamiccontext.add("id", processID);
14     event.dynamiccontext.add("publisher", getParentProcessID());
15     event.dynamiccontext.add("stacktrace", getStackTrace());
16     EventReactor.publish(event);
17 }
18 void killMPCore(){
19     ...
20     //kill the child process
21     //announce the killed event
22     ...
23 }
24 }
```

Listing 7.14: An excerpt of the class *MPCore*

The method `killMPCore` is defined likewise to destroy the child process. This method must be invoked by the media player software when the process much be killed by the software. If a child process terminates abnormally for example due to an exception in the software, the operating system notifies the parent process. For this case, the media player software must also be changed such that the parent process also publishes the event *killed* for the corresponding child process.

[3]In the EventReactor language, we assume that the active stack trace is available to be set as the dynamic contextual information of events. If the programming language does not provide the stack trace, this feature must be provided by the programmers.

7.5.3 Declaring the Publishers of Events

For each user-defined events, we must also provide the necessary information about its publisher for the EventReactor language. This information is also represented as Prolog facts.

For example the method `initMPCore` defined in the class `MPCore` in Listing 7.14 is publisher of the event *initiated* for the process *MPCore*. Listing 7.15 shows an excerpt of the code to define this method in the EventReactor language. The other publisher methods are defined similarly.

```
1 ...
2 PublisherFacts fact = new PublisherFacts();
3 fact = "isMethodWithName ('m1', 'initMPCore').
4           isClassWithName ('p1', 'MPCore').
5           isDefinedIn('m1', 'p1').";
6 EventReactor.define (fact);
7 ...
```

Listing 7.15: Declaring a publisher

7.5.4 Declaring the Reactor Types

We provide two reactor types *React* and *RestartProcess* for implementing Recoverable Process in the EventReactor language.

React is a read-only reactor type, whose only function is to publish a new event when it receives an event to process. The name of the new event may be provided as an argument to the reactor type. Otherwise the new event has the same name as the event being processes.

RestartProcess is a read-write reactor type, which restarts the specified child processes. For this matter, it access the information about the methods that kill and initializes child processes. These methods are defined in the base software, as it is explained in Section 7.5.2. This reactor type publishes the event `restarted` if it successfully restarts a process; otherwise it publishes the event `notrestarted`. These events have a parameter named as `processName`, which specifies the process that the reactor type aims to restart.

7.5.5 Representing the Concept *AppProcess*

Listing 7.16 defines the reactor chain `AppProcess_REChain` to implement the functionality of the concept *AppProcess* of the Recoverable Process technique.

The reactor chain receives four parameters. The parameters `?pinit` and `?pkill` represent the methods that create or kill a child process, respectively. The parameters `?pinited` and `?pkilled` represent the events indicating that a child process is initiated or killed, respectively.

As Figure 7.1 shows, the concept *AppProcess* of Recoverable Process has a set of attributes and events. These attributes and events are represented via the attributes and reactors in the reactor chain, respectively.

```
1  reactorchain AppProcess_REChain (?pinit,?pkill,?pinited, ?pkilled) {
2      internals
3          init : Method = ?pinit;
4          kill : Method = ?pkill;
5          pid : Integer;
6          status : String;
7
8      reactors
9          initiated_REC : React = (event == [?pinited]) = {
10          status = 'running';
11          pid = event.id;
12          name = 'inited';
13         };
14         killed_REC : React = (event == [?pkilled]) = {
15             status = 'terminated';
16             pid = −1;
17             name = 'killed';
18         };
19 }
```

Listing 7.16: A reactor chain implementing the concept *AppProcesss*

In lines 3 and 4, the attributes `init` and `kill` are defined of type `Method`, which are initialized with the parameters `?pinit` and `?pkill`, respectively. These attributes are later on used to retrieve information about the methods that kill and re-initialize the corresponding child processes.

Line 5 defines the attribute `pid` of type `Integer`, which maintains the unique identifier of the corresponding child process.

Line 6 defines the attribute `status`, which maintains the state of the corresponding child process.

Lines 9 to 13 define the reactor `initiated_REC` of type `React`.

Line 9 makes use of the keyword `event` to specify that the events represented by `?pinited` are of interest. Lines 10 to 12 define the body of the reactor.

Line 10 assigns the value `'running'` to the attribute `status` of the reactor chain.

Line 11 retrieves the unique identifier of the created process from the attribute `id` of the selected event, and assigns it to the attribute `pid` of the reactor chain.

Line 12 specifies the value `'inited'` as the name of the event that will be published by the reactor. As a result, upon the occurrence of the event `?pinited`, the event `inited` is published by the reactor, and the specified values are assigned to the attributes of the reactor chain.

Lines 14 to 18 define the reactor `killed_REC` of type `React`, which specifies the event `?pkilled` as of interest. As it is specified in the body of the reactor, upon the occurrence of `?pkilled`, the value `'terminated'` is assigned to the attribute `status`, the value `-1` is assigned to the attribute `pid`, and the value `'killed'` is specified as the name of the event that will be published by the reactor.

In the following we would like to make use of the reactor chain `AppProcess_REChain` to represent the child processes of the media player software.

Listing 7.17 defines the event package `MediaPlayerProcesses` to represent the child processes of the media player software. Lines 3 to 6 select the user-defined event `initiated` that represents the initialization of the process `MPCore`. This event is defined in the EventReactor language in Listing 7.13. Likewise, lines 8 to 11 select the user-defined event `killed`, which represents the termination of the process `MPCore`.

Lines 13 to 16 select the method `initMPCore` whose execution in the media player software initializes the process `MPCore`. This method is defined in class `MPCore` in Listing 7.14. Likewise, lines 18 to 21 select the method `killMPCore` whose execution in the media player software terminates the process `MPCore`.

Lines 24 to 29 define the event module `MPCoreProcess`, which represents the process *MPCore*. The events selected by `inited_event` and `killed_event` form the input interface of the event module. The reactor chain `AppProcess_REChain` is bound to this event module in a singleton manner, because there is only one process executing as *MPCore*. The selected events and methods are passed to the reactor chain as arguments.

At runtime when the child process *MPCore* is created in the media player software, the event `initiated` is detected in the software and is announced to the runtime environment of the EventReactor language. The announcement is performed by

the code represented in Listing 7.14. Consequently, an instance of the reactor chain `AppProcess_REChain` is created, and the reactor `initiated_REC` processes the event. After initializing the attributes of the reactor chain, the reactor publishes the event `inited`. The other child processes of interest are specified in the event package `MediaPlayerProcesses` in a similar way.

```
1  eventpackage MediaPlayerProcesses {
2      selectors
3          inited_event = {E |
4                              isEventWithName(E, 'initiated'),
5                              isProcessWithName (P, 'MPCore'),
6                              isPublishedBy(E, P)};
7
8          killed_event = {E |
9                              isEventWithName(E, 'killed'),
10                             isProcessWithName (P, 'MPCore'),
11                             isPublishedBy(E, P)};
12
13         init_method = {M |
14                             isMethodWithName (M, 'initMPCore'),
15                             isClassWithName (C, 'MPCore'),
16                             isDefinedIn(M, C)};
17
18         kill_method = {M |
19                             isMethodWithName (M, 'killMPCore'),
20                             isClassWithName (P, 'MPCore'),
21                             isDefinedIn(M, P)};
22         ...
23 eventmodules
24     MPCoreProcess := {inited_event, killed_event } <-
25                                                     singleton
26                                                     AppProcess_REChain (
27                                                             init_method, kill_method,
28                                                             init_event, kill_event
29                                                     );
30     UserInterfaceProcess := ...
31     AudioProcess := ...
32     VideoProcess := ...
33 }
```

Listing 7.17: An event module representing the child processes of interest

7.5.6 Representing the Concept *RecoveryUnit*

Listing 7.18 defines the reactor chain `RecoveryUnit_REChain` to implement the functionality of the concept *RecoveryUnit* of the Recoverable Process technique.

The reactor chain receives the list of processes that form a recovery unit via the parameter `??processes`. In the part `internals`, the attribute `processes` is defined, and is initialized with `??processes`.

In the part `reactors`, the reactor `reportFailure_REC` is defined of type `React`. Line 7 specifies `'failure'` as the name of the event that will be published by `reportFailure_REC`.

```
1 reactorchain RecoveryUnit_REChain (??processes) {
2     internals
3         processes : List = ??processes;
4
5     reactors
6         reportFailure_REC : React {
7             name = 'failure';
8         };
9 }
```

Listing 7.18: A reactor chain implementing the concept *RecoveryUnit*

Listing 7.19 defines the event package `MediaPlayerGlobalRecoveryUnit`, which defines a recovery unit for the global recovery of the media player software. In this example, we assume that the failure of the process *MPCore* is regarded as the failure of the recovery unit. Lines 3 to 6 select the event `killed`, which is published by the event module `MPCoreProcess`, indicating that the process *MPCore* is killed. The selected event is represented by the variable `mpcoreKilled_event` in the event package.

Lines 8 and 9 of Listing 7.19 select the event module `MPCoreProcess`. Likewise, lines 10 to 15 select other event modules representing the child processes of the media player software.

Lines 18 to 24 define the event module `globalRecoveryUnit`, which represents a recovery unit for the global recovery of the media player software. The event `mpcoreKilled_event` is specified as the input interface, `RecoveryUnit_REChain` is bound to it in a singleton manner because in this example there is only one instance of the event module *MPCoreProcess*, which publishes the events of interest. Therefore, there is no need to distinguish between publishers. All the selected processes are passed as the argument to the reactor chain `RecoveryUnit_REChain` .

At runtime if the process *MPCore* is killed, the event `killed` is published by the event module `MPCoreProcess`, and is selected as `mpcoreKilled_event` in Listing 7.19. The event is provided to the instance of `RecoveryUnit_REChain` that is bound to the event module `globalRecoveryUnit`. The reactor `notifyFailure_REC`

defined in the reactor chain `RecoveryUnit_REChain` processes the event, and publishes the event `failure`, as it is specified in line 7 of Listing 7.18.

```
1  eventpackage MediaPlayerGlobalRecoveryUnit{
2      selectors
3          mpcoreKilled_event = {E |
4                                  isEventWithName(E, 'killed'),
5                                  isEventModuleWithName (EM, '*.MPCoreProcess'),
6                                  isPublishedBy(E, EM)};
7
8          mpcoreProcess = {EM |
9                                  isEventModuleWithName (EM, '*.MPCoreProcess')};
10         uiProcess = {EM |
11                                 isEventModuleWithName (EM, '*.UserInterfaceProcess')};
12         audioProcess = {EM |
13                                 isEventModuleWithName (EM, '*.AudioProcess')};
14         videoProcess = {EM |
15                                 isEventModuleWithName (EM, '*.VideoProcess')};
16
17     eventmodules
18         globalRecoveryUnit := {mpcoreKilled_event} <-
19                                                 singleton
20                                                 RecoveryUnit_REChain({mpcoreprocess,
21                                                                       uiProcess,
22                                                                       audioProcess,
23                                                                       videoProcess
24                                                                      });
25
26 }
```

Listing 7.19: An event module representing the global recovery unit

Likewise, Listing 7.20 defines a recovery unit for the local recovery of the process *UserInterface*.

```
1  eventpackage MediaPlayerLocalRecoveryUnit{
2      selectors
3          uiKilled_event = {E |
4                              isEventWithName(E, 'killed'),
5                              isEventModuleWithName (EM, '*.UserInterfaceProcess'),
6                              isPublishedBy(E, EM)};
7
8          uiProcess = {EM |
9                              isEventModuleWithName (EM, '*.UserInterfaceProcess')};
10
11     eventmodules
12         uiLocalRecoveryUnit := {uiKilled_event} <-
13                                                    singleton
14                                                    RecoveryUnit_REChain({uiProcess});
15 }
```

Listing 7.20: An event module representing a local recovery unit

7.5.7 Representing the Concept *ProcessManager*

Listing 7.21 defines the reactor chain `ProcessManager_REChain` to implement the functionality of the concept *ProcessManager* of the Recoverable Process technique.

Line 3 defines the reactor `recovery_REC` of user-defined type `RestartProcess`. The type can process the events that are published by the instances of the reactor chain `RecoveryUnit_REChain`. It retrieves the list of processes to be recovered from the attribute `processes` of the corresponding instance of `RecoveryUnit_REChain`. Each process is actually represented by an instances of `AppProcess_REChain`. The methods that kill and re-initialize a process are defined in the attributes `init` and `kill` of `AppProcess_REChain`. The reactor type `RestartProcess` invokes these methods for each process to re-initialize it.

At runtime, if the event `failure` is published by any of the specified recovery units, the reactor `recovery_REC` is informed of the event, and restarts the processing forming the recovery unit.

```
1  reactorchain ProcessManager_REChain{
2      reactors
3          recovery_REC : RestartProcess;
4  }
```

Listing 7.21: A reactor chain implementing the concept *ProcessManager*

Listing 7.22 defines the event package `MediaPlayerProcessManagers`. Lines 3 to 6 select the event `failure` that is published by the event module `globalRecoveryUnit`, and represent it via the variable `globalFailure_event` in the event package.

```
1  eventpackage MediaPlayerProcessManagers{
2      selectors
3          globalFailure_event = {E |
4                                  isEventWithName(E, 'failure'),
5                                  isEventModuleWithName (EM, '*.globalRecoveryUnit'),
6                                  isPublishedBy(E, EM)};
7
8          localFailure_event = {E |
9                                 isEventWithName(E, 'failure',
10                                isEventModuleWithName (EM, '*.uiLocalRecoveryUnit'),
11                                isPublishedBy(E, EM)};
12
13     eventmodules
14         globalRecovery := {globalFailure_event} <- singleton ProcessManager_REChain;
15
16         uiLocalRecovery := {localFailure_event} <- singleton ProcessManager_REChain;
17 }
```

Listing 7.22: Event modules representing process managers

Here, lines 8 to 11 select the event `failure` that is published by the event module `uiLocalRecoveryUnit`, and represent it via the variable `localFailure_event` in the event package.

Line 14 defines the event module `globalRecovery`, specifies the event selected by `globalFailure_event` as its input interface, and binds `ProcessManager_REChain` to it in a singleton manner, because there is only one instance of the event modules `globalRecoveryUnit` and `uiLocalRecoveryUnit`. Likewise, the event module `uiLocalRecovery` is defined in line 16.

7.5.8 Representing Recovery Constraints

Listing 7.23 defines the event package `RecoveryConstraint`, which specifies a constraint between the global recovery of the media player software and the local recovery of the process *UserInterface*.

Lines 3 and 5 select the event modules `globalRecovery` and `uiLocalRecovery`, which are defined in Listing 7.22. The selected event modules are represented them via variables `globalrecovery_em` and `uilocalrecovery_em` in the event package `RecoveryConstraint`.

Line 8 specifies that the event module represented by `uilocalrecovery_em` must ignore the events that are published during the execution `globalrecovery_em`.

```
1 eventpackage RecoveryConstraint{
2     selectors
3         globalrecovery_em = {EM | isEventModuleWithName (EM, '*.globalRecovery')};
4
5         uilocalrecovery_em = {EM | isEventModuleWithName (EM, '*.uiLocalRecovery')};
6
7     constraints
8         ignore(uilocalrecovery_em, globalrecovery_em);
9 }
```

Listing 7.23: Representing recovery constraints

More complex composition constraints can be implemented via dedicated reactor types. For example, assume that if the global recovery fails to re-initialize *UserInterface*, the local recovery must try to do so. As it is explained in Section 7.5.4, the reactor type `RestartProcess` publishes the event `restarted` if it successfully restarts a process; otherwise it publishes the event `notrestarted`. These events have a parameter named as `processName`, which specifies the process that `RestartProcess` aimed to restart.

Listing 7.24 defines the reactor chain `Coordinator_REChain`, which receives two parameters. The parameter `?failedProcess` represents the name of a process that was failed to be re-initialized. The process is represented by the parameter `??processes`.

Lines 3 to 5 define the reactor `recovery_secondtry` of type `RestartProcess`. The reactor selects those events whose attribute `processName` equals to `?failedProcess`. In the body of the reactor, the parameter `??processes` is assigned to the parameter `processes` of the reactor.

```
1 reactorchain Coordinator_REChain (?failedProcess, ??processes) {
2  reactors
3     recovery_secondtry : RestartProcess = (event.processName ==?failedProcess){
4         processes = ??processes;
5     }
6 }
```

Listing 7.24: A reactor chain implementing an application-specific composition constraint

Listing 7.25 defines the event package `Coordination`. Lines 3 to 6 select the event `notrestarted` that is published by the event module `globalRecovery`. Line 8 selects the event module `UserInterfaceProcess`.

Lines 11 and 12 define the event module `coordinator`, which specifies the event represented by the variable `failure_event` as its input interface. The reactor chain `Coordinator_REChain` is bound to this event module in a singleton manner, and the selected process and its name are passed as its argument.

As a result, if the event module `globalRecovery` publishes the event `notrestarted` for the process *UserInterface*, the event module `coordinator` will be instantiated, and the reactor chain `Coordinator_REChain` will restart the process.

```
1  eventpackage Coordination{
2      selectors
3          failure_event = {E |
4                               isEventWithName (E, 'notrestarted'),
5                               isEventModuleWithName (EM, '*.globalRecovery'),
6                               isPublishedBy( E, EM)};
7
8          uiProcess = {U | isEventModuleWithName (EM, '*.UserInterfaceProcess')};
9
10     eventmodules
11         coordinator := {failure_event} <- singleton
12                                           Coordinator_REChain('UserInterface', {uiProcess});
13 }
```

Listing 7.25: An event module representing an application-specific composition constraint

7.5.9 Evaluation

An evaluation of the EventReactor language with respect to its support for Event Composition Model is provided in Chapter 4. In the following, we make use of the implementation provided for Recoverable Process to discuss the suitability of EventReactor in providing a natural representation for the concepts of interest.

As the listings in previous subsections show, because of implementing Event Composition Model, natural representation of concepts can be achieved in EventReactor. The open ended-ness with new events enables us to define process-related events to the language and consequently, represent the concept *AppProcess*.

Open ended-ness with new reactor types facilitates defining the behavior of various different concepts and application-specific composition constraints. Parameterizable reactor chains enables us to provide abstract and reusable implementations for the concepts; where the application-specific representations of the concepts are provided via event modules.

The flexibility of binding reactor chains to event modules improves the modularity of events modules and reactor chains. Since reactors can publish events, and dedicated Prolog expressions are provided to select such events, we can compose event modules with each other in a hierarchal manner. This improves the modularity and compose-ability of the event modules. In addition, the abstraction level of event modules increases as their position in the hierarchy elevates. For example, the event module representing the concept *RecoveryUnit* is an abstraction over the processes that belong to a recovery unit. The abstraction level increases further due to the language-independency of the EventReactor language.

As Listing 7.23 shows, constraints can be defined among event modules. Such constraints can be modularized in separate event packages, without influencing the specification of event modules. This helps to increase the reusability of event modules.

7.6 Conclusion

This chapter makes use of a runtime enforcement technique called Recoverable Process to illustrate the shortcomings of existing runtime enforcement frameworks and their underlying languages in providing natural representation for the concepts of a process recovery technique.

The chapter provides an implementation of Recoverable Process in the EventReactor language. It shows that the support for Event Composition Model by the EventReactor language enables us to represent the concepts of Recoverable Process naturally.

Chapter 8

Conclusion and Future Work

Due to the ever increasing complexity of software, it may not be possible to detect all potential failures before the actual execution of the software in its target environment. Therefore, runtime enforcement techniques are introduced to allow the software continuing its operation in case of failures.

Runtime enforcement techniques check the actual execution of software against the formally specified properties of the software. If a failure is detected, diagnosis and recovery actions may be performed to detect the causes of the failures and to recover the software from the failure, respectively.

There is an increasing number of runtime enforcement frameworks in the literature. These frameworks offer specification languages to express the properties, diagnosis rules and recovery strategies. They also offer a compiler that generates code from the specifications. This code can be expressed in the same language as the base software or it can be in an intermediate language, which abstracts the base software. The generated code is integrated with base software from the perspective of runtime enforcement.

8.1 Problem

In the thesis, the term underlying language is used to refer to the language of the generated code. To ease the code generation and integration, specification languages usually adopt the elements of their underlying languages.

Despite the practical advantages of this approach, the thesis identifies that it may not be possible to represent runtime enforcement concepts *naturally*, if the elements of the underlying language are adopted by specification languages. Consequently, we must provide workaround representations for the concepts of interest using the elements available in specification languages. The thesis identifies the following three shortcomings in such workarounds:

- **Decreased modularity**: The representation of a concept can be scattered across multiple underlying language elements due to not directly representing the concepts of interest. Moreover, the representation of a concept may also get tangled with the representation of other concepts.
- **Decreased abstraction level**: The abstraction level of specifications may be too close to the level of code in underlying languages. Consequently, the specifications cannot be reused for software developed in different languages and platforms.
- **Decreased compose-ability**: The composition of concepts cannot also be represented naturally. Therefore, workarounds must be provided for them, and such workarounds also suffer from the decreased modularity, decreased abstraction level and decreased compose-ability.

To overcome these shortcomings, the thesis discusses that specification languages must offer first-class abstractions that correspond one-to-one to the runtime enforcement concepts. These abstractions must not incorporate the implementation languages and process structure. Specification languages must also offer a rich set of constructs to implement variable composition strategies.

The thesis evaluates a representative set of runtime enforcement frameworks with respect to their support for the above-mentioned requirements. It identifies that none of these frameworks completely fulfill these requirements, so they fall short in providing natural representation for the concepts of interest.

8.2 Solution

To facilitates a natural representation of concepts, the thesis identifies the characteristic features of runtime enforcement techniques, and accordingly proposes a computation model named as Event Composition Model that respect these features. The thesis introduces the EventReactor language as an implementation of this computation model.

8.2.1 Characteristic Features of Runtime Enforcement Techniques

To propose a solution to the identified problems, chapter 2 of the thesis identifies that the existing runtime enforcement techniques have the following characteristic features, which must be taken into account by specification languages:

- **Transient nature of runtime enforcement concepts:** The interactions among the concepts of runtime enforcement techniques have by nature a transient characteristic. Here, the changes to the states of a concept may derive other concepts. Therefore, specification languages must provide elements that represent the changes in the states of interest in each concept.
- **Open ended-ness of the kinds of elements in specifications:** It may be required to represent various different kinds of concepts in the specifications. Functions, objects, processes, subsystems are examples. It is not easy or even possible to foresee all kinds of elements that are desired to be represented in the specification languages. Therefore, specification languages and their implementations must also be open-ended with respect to their elements.
- **No strict hierarchy among specifications:** In general, there is no strict hierarchy among the specifications. This implies that, for example, a system with diagnosis and recovery processes may be considered as base software as well. This results in multi-levels of runtime enforcement concepts. The thesis claims that specification languages must facilitate arbitrary composition of concepts with each other.

8.2.2 Event Composition Model

Chapter 3 of the thesis proposes Event Composition Model, which is a computation model for specification languages of runtime enforcement frameworks. Event Composition Model offers a set of elements that respect the previously-mentioned characteristic features of runtime enforcement techniques.

Event Composition Model considers the notion of event suitable to represent the transient nature of concepts. However, event is a too low-level of a representation with respect to the concerns of interest in specifications.

Event Composition Model offers the notion of event modules, which abstracts over a group of correlated events. The selection of group of events is defined by an event composition language, which is capable of selecting any event that is declared in

the system and is in the scope. The composition language is expressive enough to select events of interest that occur in programs implemented in different languages, running on different processors and/or hosts.

An event module has input interface, output interface and a set of implementations that are termed reactors. The set of events that are grouped by an event module form the input interface of the module. Reactors may publish new events. These are regarded as the output interface of the event module, and can be selected by the event composition language. If multiple reactors process an event, their applicability to the event can be constrained via an event constraint language.

Reactors have type, and two kinds of types are distinguished: Read-only and Read-write. In contrast to the read-write reactors, the read-only reactors cannot modify the states of software, so they do not have functional side-effects on software. Some concepts such as verification and diagnosis have read-only nature, meaning that they only gather information from other concepts and reason about the information without modifying the information. Some other concepts such as recovery have read-write nature, because they modify other concepts. The distinction between read-only and read-write reactor types helps to represent the concepts more naturally.

Event Composition Model is extendable with new kinds of events, event modules and reactors. This enables us to introduce new kinds of elements in specifications. The event composition language facilitates selecting events published by the implementations of event modules, and grouping them to define new event modules. This helps to form arbitrary hierarchies of event modules.

8.2.3 The EventReactor Language

The thesis evaluates a representative set of programming languages with respect to their support for Event Composition Model. The evaluation reveals that these languages do not fully support the computation model.

Chapter 4 of the thesis introduces the EventReactor language, as an implementation of Event Composition Model. The EventReactor language has the following features:

- It supports a set of predefined events in programs, and is extendable with new user-defined events. It provides an API to publishers to announce the occurrence of the user-defined events.
- It utilizes the Prolog language as its event composition language. The predefined and user-defined events are defined in the language via a set of Prolog facts, which provide sufficient information to select them.

- It offers a linguistic construct to define event modules, which groups a set of selected events, binds a set of reactors to it and specifies the instantiation strategy of the event module.
- It facilitates defining and instantiating reactor types, which may also publish new events. Dedicated Prolog expressions are provided to select such events.
- It supports the notion of reactor chains, which is a module consisting a set of reactors that are composed with each other to process events in sequence.
- It provides linguistic constructs to compose event modules, according to specific constraints.
- It enables defining dedicated composition constraints via reactor types.
- It modularizes the specifications of event modules, reactors, reactor chains and compositions.
- Its linguistic constructs are independent of the implementation languages of the base software. Its compiler supports Java, .Net and C languages. This feature of the EventReactor language facilitates reusing specifications for software developed in various languages. This is illustrated in Chapter 6.
- For distributed Java base software that makes use of RMI as middleware, its linguistic constructs are transparent from the process structures of the base software. For multiple-language base software, programmers must specify the distribution information for the specification of event module. This is illustrated in Chapter 6.

8.3 Evaluation of the EventReactor Language

The thesis makes use of a runtime enforcement technique named Recoverable Process to evaluate the suitability of the EventReactor language in representing the concepts of interest naturally. As it is defined in Chapter 2 of the thesis, by the term naturally, we mean a representation that is modular, is at the right abstraction-level without incorporating implementation context, and is compose-able with the representations of other concepts.

Recoverable Process aims at making processes fault-tolerant, by monitoring the processes, detecting their failures and restarting a failed process along with the other processes that are semantically related to it. The core concepts in Recoverable Process are: the processes of interest, a group of semantically related processes, which

is named as *RecoveryUnit*, *ProcessManager* that implements the functionality to recover the processes, and *CommunicationManager* that implements the functionality to recover inter-process messages.

In the literature, an implementation of Recoverable Process in the C language is available. Here, *ProcessManager* and *CommunicationManager* are implemented as separate modules. These modules also maintain information about the processes that must be recovered and their memberships in the recovery units. These modules are invoked from within the modules of the base software.

In Chapter 7, the thesis explains that the C-based implementation of Recoverable Process falls short in representing the concepts of Recoverable Process naturally, because of the following three reasons. First, there is no explicit and modular representation of the processes of interest and recovery units; their definition is tangled in *ProcessManager* and *CommunicationManager*. Second, the invocations to *ProcessManager* and *CommunicationManager* are scattered across and tangled with the modules in the base software. Third, the abstraction level of the representations is at the code level, because they are represented via the C language.

To overcome the above-mentioned shortcomings, Chapter 7 explains a possible implementation of Recoverable Process in the JavaMOP language. JavaMOP facilitates representing the concepts of interest in specification modules separately from the base software. However, the composition mechanism of JavaMOP does not facilitate composing individual specification modules with each other. As a result, we have to sacrifice the modularity of the concepts *RecoveryUnit* and *ProcessManager* and represent them in one specification module. Moreover, some of the concepts of interest, such as processes, cannot directly be represented in JavaMOP. Finally, the abstraction level of representations is at the code level, because they incorporate the elements of the Java language.

The thesis discusses a possible implementation of Recoverable Process in the AspectJ language. A modular representation of processes, *RecoveryUnit* and *ProcessManager* is provided via abstract aspects. The application of these concepts to base software is represented via concrete aspects. To some extent, this helps to achieve a modular representation for the concepts of interest. The thesis illustrates that the composition of concepts with each other cannot be modularized as it is desired, and changes in composition constraints may impose major changes on the existing aspects. Moreover, the abstraction level of the representations is at the code level.

In Chapter 7, the thesis illustrates that natural representation of the concepts of Recoverable Process can be achieved in the EventReactor language. For this matter, the set of process-related events and their publishers are defined in the EventReactor

language. Dedicated reactor types are defined to implement the functionality of the concepts.

The concepts are represented via event modules. A reusable implementation of the event modules is provided via reactor chains, which are defined independently from the implementation language and distribution of the base software.

The distinction between read-only and read-write reactor types in the EventReactor language helps to preserve natural characteristics of the concepts. For example, processes of interest and *RecoveryUnit* are represented via a set of read-only reactor types. Whereas process manager is represented via read-write reactor types, because it has side-effects on the base software, i.e. restarting the processes.

The modularity and compose-ability of the concepts increases due to the possibility to compose event modules with each other in a hierarchal manner. For example in the implementation of Recoverable Process, the processes of interest are represented via event modules that reside at the lowest level of the hierarchy, and *RecoveryUnit* is an event module that resides at one level higher and composes the event modules representing the processes. *ProcessManager* is another event module that is composed with *RecoveryUnit*, and so on. Chapter 7 shows that Application-specific composition constraints are implemented via dedicated reactor types, and are represented as event modules that apply to other event modules of interest.

The EventReactor language is declarative and various checks is provided by its compiler to ensure the correctness of the specifications.

8.4 Implementation Challenges

The main challenge that we face is to manage instances of event modules for a group of correlated publishers. We make use of relational tables to maintain information about the groups of correlated publishers and the corresponding instances of event modules. This enables us to benefit from the features of the relational tables in various ways.

As our experiments in Chapter 6 reveal, the size of the table can easily grow if the number of publishers or threads of execution increases. As a result, the time to look up information in the table and the required memory space to maintain the table increase. If the publishers are distributed across multiple processes, the table is maintained in a separate process. This imposes inter-process communication, which increases the time to access information in the table further.

We still need to investigate more along this line to reduce the size of the tables and to increase the speed of the operations that are performed on the tables.

A possible solution to reduce the size of the tables is to specify constraints for the correlated publishers. A typical example of such constraints in the runtime enforcement literature is the Java *Collection* and *Iterator* classes. It is said in the literature that linguistic constructs must be offered for selecting *Collection* and correlated *Iterator* objects effectively.

The current version of the EventReactor language only facilitates grouping the correlated publishers, without the possibility to constrain them. Nevertheless, we believe that desired constraints can be expressed as Prolog expressions, and can be added to the language. For example, by the help of Prolog expressions, it is possible to select the classes that have inheritance correlation among each other. One may also adopt the expressions that select the classes whose instances are reachable from the instances of a specified class.

The implementation of such constraints is feasible, due to the use of relational tables. It is possible to define a set of conditions for each column of a relational table, as an access control constraint; only if the conditions are satisfied, access can be allowed. Since event module tables are also relational tables, a specified correlation constraint can be defined as the condition that publishers must satisfy before the tables are modified.

To reduce the required time to look up in the tables, we may consider investigating on the other alternatives mentioned in Chapter 5. For example, if all the selected events of interest are announced by one publisher, it will be more efficient to keep a direct link between the publisher and the corresponding instances of event modules.

8.5 Future Work

As future work, we consider utilizing the EventReactor language in various different domains, whose concepts have the same characteristic features as the runtime enforcement domain.

The runtime environment of the EventReactor language can be improved in various ways. Currently, information about events and event modules are maintained in shared data storage. Upon the occurrence of an event, the runtime environment accesses the shared data storage, and retrieves the information about the event, corresponding event modules and reactor chains. To eliminate or reduce the look up

time, it may be possible to inline this information in the program code. We consider investigating more along this line.

Another direction is the distribution of event modules and reactor chains across multiple processes. We are interested to investigate on the other alternatives in sharing event module tables among multiple processes, and measure their runtime overhead. We are also interested in considering the possible link between the process structure of the base software, the degree to which the specified events are distributed, and the runtime overhead.

Providing more linguistic constructs to allow programmers specify the process structure of the base software, and to specify the process in which the instances of event modules and reactor chains must be maintained is another future line of research.

We also consider extending EventReactor with various compile-time checks. An example is to check if multiple read-write reactors have side-effects on a shared resource, and to ensure that the effects are not conflicting.

Bibliography

[1] Dictionary.com, http://dictionary.reference.com/browse/event.

[2] Regexp, http://swtch.com/ rsc/regexp/dfa0.c.txt.

[3] Yevgenia Alperin-Tsimerman and Shmuel Katz. Dataflow Analysis for Properties of Aspect Systems. In *5th Haifa Verification Conference (HVC)*, LNCS, 2009.

[4] AspectC. http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[5] AspectWerkz. http://aspectwerkz.codehaus.org.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[7] David A.Watt. *Programming Language Design Concepts*. Wiley, 2004.

[8] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. pages 49–69. Springer, 2004.

[9] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.

[10] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with Assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):1–15, 2001.

[11] Lujo Bauer, Jay Ligatti, and David Walker. Composing Security Policies with Polymer. *SIGPLAN Not.*, 40:305–314, June 2005.

[12] Lujo Bauer, Jay Ligatti, and David Walker. Composing Expressive Runtime Security Policies. *ACM Trans. Softw. Eng. Methodol.*, 18, 2009.

[13] Lodewijk Bergmans, Mehmet Aksit, and Bedir Tekinerdogan. Aspect Composition using Composition Filters. In M. Aksit, editor, *Software Architectures and Component Technology*, International Series in Engineering and Computer Science, pages 357–384. Kluwer Academic Publishers, 2002.

[14] Christoph Bockisch, Somayeh Malakuti, Shmuel Katz, and Mehmet Aksit. Making Aspects Natural: Events and Composition. In *Proceedings of the 10th international conference on Aspect-oriented software development (modularity vision track)*, AOSD, pages 285–299, Porto de Galinhas, Brazil, 2011. ACM.

[15] Microsoft Corporation. C# language specification. http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx.

[16] Patrice Chalin, Joseph Kiniry, Gary Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer Berlin / Heidelberg, 2006.

[17] Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM press, 2007.

[18] Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (jml). In *Proceedings Of The International Conference On Software Engineering Research And Practice (Serp 02), Las Vegas*, pages 322–328. CSREA Press, 2002.

[19] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.

[20] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML - Progress and Issues in Building and Using ESC/Java2. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*. SpringerVerlag, 2004.

[21] Compose*. http://composestar.sourceforge.net/.

[22] CORBA. http://www.corba.org.

[23] T.H. Cormen, C.E. Leiserson, R.L. Riyest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[24] N. Delgado, A.Q. Gates, and S. Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *Software Engineering, IEEE Transactions on*, 30(12):859 – 872, 2004.

[25] Dhananjay.M. Dhamdhere. *Operating Systems: A Concept-Based Approach*. McGraw Hill Higher Education, 2006.

[26] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[27] Arvind Easwaran, Sampath Kannan, and Oleg Sokolsky. Steering of Discrete Event Systems: Control Theory Approach. *Electron. Notes Theor. Comput. Sci.*, 144:21–39, May 2006.

[28] Patrick Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded Contract Languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, Sierre, Switzerland, 2010. ACM.

[30] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, PADD '88, pages 183–194, Madison, Wisconsin, United States, 1988. ACM.

[31] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2004.

[32] Robert G. Freeman and Arup Nanda. *Oracle Database 11g New Features*. McGraw-Hill Osborne Media, 2007.

[33] Jeffrey E.F. Friedl. *Mastering Regular Expressions*. O'Reilly Media, 2006.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[35] Vaidas Gasiu nas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. Declarative Events for Object-Oriented Programming. Research Report RR-7313, INRIA, 05 2010.

[36] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular Event-Driven Object Interactions in Scala. In *Proceedings of the 10th international conference on Aspect-oriented software development*, AOSD, pages 227–240, Porto de Galinhas, Brazil, 2011. ACM.

[37] Bruno Harbulot and John R. Gurd. A Join Point for Loops in AspectJ. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD, pages 63–74, Bonn, Germany, 2006. ACM.

[38] Klaus Havelund. Runtime Verification of C Programs. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 7–22. Springer Berlin / Heidelberg, 2008.

[39] Linda G. Hayes. *Automated Testing Handbook*. Software Testing Inst., 2004.

[40] Michiel Hendriks. Compose* Annotated Reference Manual. 2007.

[41] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21:666–677, August 1978.

[42] Kevin Hoffman and Patrick Eugster. Cooperative Aspect-Oriented Programming. *Sci. Comput. Program.*, 74:333–354, March 2009.

[43] J. E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.

[44] Java. http://www.oracle.com/technetwork/java/index.html.

[45] Java-JNI. http://download.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html.

[46] Java RMI. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

[47] JavaMOP. http://fsl.cs.uiuc.edu/index.php/JavaMOP_Syntax.

[48] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring Distributed Systems. *ACM Trans. Comput. Syst.*, 5:121–150, March 1987.

[49] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[50] JVM Tool Interface. http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html.

[51] Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*. Wiley, 1999.

[52] Shmuel Katz. Aspect Categories and Classes of Temporal Properties. In *TAOSD*, LNCS, pages 106–134. 2006.

[53] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[54] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[55] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design*, 24:129–155, 2004.

[56] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications.

[57] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis using SPARK. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[58] Karl J. Lieberherr and David Lorenz. Coupling Aspect-Oriented and Adaptive Programming. In Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. In press.

[59] Linux. http://www.linux.org.

[60] Xavier Logean, Falk Dietrich, Hayk Karamyan, and Shawn Koppenhöfer. Run-Time Monitoring of Distributed Applications. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 459–474, London, UK, 1998. Springer-Verlag.

[61] Somayeh Malakuti, Mehmet Aksit, and Christoph Bockisch. Distribution Transparency in Runtime Enforcement. In *Proceedings of the 2011 IEEE/FTRA International Conference on Advanced Software Engineering*. IEEE Press, 2011.

[62] Somayeh Malakuti, Mehmet Aksit, and Christoph Bockisch. Runtime Verification in Distributed Computing. *Journal of Convergence: An International Journal of Future Technology Research Association International*, 2(1):11, 2011.

[63] Somayeh Malakuti, Christoph Bockisch, and Mehmet Aksit. Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software. In *Proceedings of the 20th IEEE international conference on software reliability engineering*, ISSRE'09, pages 31–40, Piscataway, NJ, USA, 2009. IEEE Press.

[64] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. *SIGPLAN Not.*, 40:365–383, October 2005.

[65] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*. 1989.

[66] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient Monitoring of Parametric Context-Free Patterns. *Journal of Automated Software Engineering*, 17(2):149–180, June 2010.

[67] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.

[68] Russ Miles and Kim Hamilton. *Learning UML 2.0*. Manning Publications, 2006.

[69] Richard Mitchell and Jim McKim. *Design by Contract, by Example*. Addison-Wesley Professional, 2001.

[70] MPlayer. http://www.mplayerhq.hu.

[71] Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt. Debugging and Testing Middleware with Aspect-Based Control-Flow and Causal Patterns. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 183–202, Leuven, Belgium, 2008. Springer-Verlag.

[72] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2004.

[73] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD, pages 7–15, Lancaster, UK, 2004. ACM.

[74] Martin Odersky. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Inc, 2008.

[75] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In Mehmet Aksit, editor, *In Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001.

[76] Chris Allan Pavel, Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA*, pages 345–364, 2005.

[77] Hridesh Rajan and Gary Leavens. Ptolemy: A Language with Quantified, Typed Events. In Jan Vitek, editor, *ECOOP 2008 Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179. Springer Berlin / Heidelberg, 2008.

[78] Hridesh Rajan and Kevin Sullivan. Eos: Instance-Level Aspects for Integrated System Design. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 297–306, Helsinki, Finland, 2003. ACM.

[79] Hridesh Rajan and Kevin J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 59–68, St. Louis, MO, USA, 2005. ACM.

[80] M.R. Riordan. *Designing Relational Database Systems*. Microsoft Press, 1999.

[81] David S. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 92–104, Melbourne, Australia, 1992. ACM.

[82] Grigore Roşu, Feng Chen, and Thomas Ball. Synthesizing Monitors for Safety Properties – This Time with Calls and Returns –. In *Workshop on Runtime Verification (RV'08)*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008.

[83] B.G. Ryder. Constructing the Call Graph of a Program. *Software Engineering, IEEE Transactions on*, SE-5(3):216 – 226, May 1979.

[84] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association Aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD, pages 16–25, Lancaster, UK, 2004. ACM.

[85] Usa Sammapun and Oleg Sokolsky. Regular Expressions for Run-Time Verification. In *Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis*, 2003.

[86] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 418–427, Washington, DC, USA, 2004. IEEE Computer Society.

[87] Soot Framework. www.sable.mcgill.ca/soot/.

[88] Hasan Sozer. *Architecting Fault-Tolerant Software Systems*. PhD thesis, University of Twente, 2009.

[89] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Trans. Softw. Eng. Methodol.*, 20:1:1–1:43, July 2010.

[90] Leon Sterling and Ehud Shapiro. *The Art of Prolog, Second Edition: Advanced Programming Techniques (Logic Programming)*. The MIT Press, 1994.

[91] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 178–204, London, UK, 2002. Springer-Verlag.

[92] TRADER project, ESI, 2009. http://www.esi.nl.

[93] Danny Weyns Eddy Truyen. Distributed Threads in Java. In *In Proceedings of the International Symposium on Distributed and Parallel Computing, ISDPC*, 2002.

[94] Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful Aspects in JAsCo. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition*, LNCS, pages 167–181. 2005.

[95] Nathan Weston, Francois Taiani, and Awais Rashid. Interaction Analysis for Fault-Tolerance in Aspect-Oriented Programming. In *MeMoT*, pages 95–102, 2007.

[96] Dehua Zhang, E. Duala-Ekoko, and L. Hendren. Impact Analysis and Visualization Toolkit for Static Crosscutting in AspectJ. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 60 –69, May 2009.

[97] Jun Zhu, Changguo Guo, Quan Yin, Jianlu Bo, and Quanyuan Wu. A Runtime-Monitoring-Based Dependable Software Construction Method. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 1093 –1100, 2008.

Samenvatting

Runtime enforcement technieken zijn in de literatuur gentroduceerd om het hoofd te bieden aan fouten die optreden wanneer software wordt gexecuteerd in zijn doelomgeving. Deze technieken bieden diagnose- en herstelacties om respectievelijk de oorzaken van de fouten te identificeren en deze te herstellen.

Aangezien het ontwikkelen van runtime enforcement technieken ingewikkeld, foutgevoelig en duur kan zijn, worden runtime enforcement frameworks voorgesteld om het ontwikkelingsproces te vergemakkelijken. Hiertoe ondersteunen deze frameworks verschillende talen om de gewenste eigenschappen van software te specificeren, en om herstelstrategien te definiren. Gebaseerd op de specificaties genereren runtime enforcement frameworks code en integreren ze deze met de software die geverifieerd of hersteld moet worden. De code wordt gewoonlijk gegenereerd in dezelfde taal als die gebruikt is om de software te implementeren, of in een tussentaal die de software abstraheert.

Ongelukkigerwijs gebruiken de specificatietalen de elementen van de programmeertalen van de gegenereerde code, en daarom schieten zij tekort om *op natuurlijke wijze* runtime enforcement concepten te representeren. Met de term concept bedoelen we een fundamentele abstractie of definitie die voorkomt in de runtime enforcement technieken. Dientengevolge kan implementatie van runtime enforcement concepten kampen met verstrooing en verwarring. Dit beperkt de modulariteit en samenstelbaarheid van de specificatie van runtime enforcement concepten. Bovendien, gebruik van de elementen van de onderliggende talen in specificaties doet de specificaties te zeer toegesneden zijn op de gebruikte programmeertalen en platformen. Dit vermindert de herbruikbaarheid en de begrijpelijkheid van de specificaties, en verhoogt hun complexiteit.

Om een natuurlijke representatie van runtime enforcement concepten mogelijk te maken introduceert dit proefschrift een berekeningsmodel, Event Composition Model

geheten, dat rekening houdt met de karakteristieke kenmerken van runtime enforcement concepten. Dit berekeningsmodel biedt een verzameling nieuwe taalabstracties, die **events**, **event modules**, **reactors**, **reactor chains**, **event composition language** en **event constraint language** worden genoemd. Events representeren veranderingen in de van belang zijnde toestanden. Event modules zijn middelen om events te groeperen, hebben input-output interfaces, en implementaties. Reactors zijn de implementaties van event modules. Reactor chains zijn groepen gerelateerde reactors die een serie events behandelen. De event composition language faciliteert het selecteren van van belang zijnde events, en de event constraint language faciliteert het definiren van constraints tussen reactors of event modules.

Een belangrijk aandachtspunt is de vraag hoe het Event Composition Model kan worden gemplementeerd met gebruikmaking van huidige programmeertalen. Hiertoe worden in dit proefschrift de relevante programmeertalen gevalueerd met betrekking tot hun ondersteuning van implementatie van het Event Composition Model. De evaluatie laat zien dat geen van de bestaande talen het model kan implementeren zodanig dat de gewenste kwaliteitsvereisten, zoals modulariteit, abstractie en samenstelbaarheid, vervuld worden. Niettemin, aspect-gerienteerde talen bieden hiertoe de meest veelbelovende eigenschappen.

Het proefschrift introduceert de taal EventReactor, als de opvolger van de aspect-gerienteerde taal Compose*, die het Event Composition Model implementeert. De taal is geschikt voor nieuwe soorten events en reactor typen. Dit ondersteunt het representeren van nieuwe soorten concepten. De taal maakt gebruik van de taal Prolog als zijn event composition language. Reactors en reactor chains zijn parametriseerbaar, en worden gescheiden van event modules gedefinieerd. Dit verhoogt de herbruikbaarheid van event modules en hun implementaties.

Hedendaagse systemen worden meer en meer gemplementeerd met gebruikmaking van meerdere talen, en deze trend lijkt ook in de nabije toekomst voort te duren. De huidige runtime enforcement frameworks schieten echter tekort in het ondersteunen van software die is gemplementeerd in verschillende talen. In het Event Composition Model maakt de event composition language het mogelijk om events te selecteren van systemen die in verscheidene talen gemplementeerd zijn. In de taal EventReactor worden specificaties onafhankelijk van enige programmeertaal gedefinieerd, en de compiler van EventReactor maakt het mogelijk code te genereren voor de talen Java, C en .Net. Bijgevolg kunnen de specificaties worden hergebruikt voor software die wordt ontwikkeld in verschillende talen.

Het is nu meer en meer gebruikelijk dat applicaties worden ontworpen om gexecuteerd te worden op gedistribueerde systeemarchitecturen. Helaas kunnen de meeste runtime enforcement frameworks niet gebruikt worden voor gedistribueerde systemen. Er zijn een paar runtime enforcement frameworks die wel kunnen werken met

gedistribueerde architecturen. Deze systemen hebben echter specificaties die informatie bevatten van de onderliggende proces-structuur. Dit verhoogt de complexiteit en vermindert de herbruikbaarheid van de specificaties als de proces-structuur van de software verandert. De taal EventReactor pakt dit probleem aan door het ondersteunen van transparantie van gedistribueerdheid.

Er zijn twee basismanieren om de taal EventReactor te gebruiken: a) als een taal die ten grondslag ligt aan de specificatietalen van runtime enforcement frameworks; b) als een implementatietaal van runtime enforcement technieken.

Dit proefschrift legt het Event Composition Model, de taal EventReactor die dit model implementeert, en de compiler van deze taal uit door middel van het gebruik van illustratieve voorbeelden. Het proefschrift maakt gebruik van een voorbeeld van een runtime enforcement techniek, Recoverable Process genaamd, om de geschiktheid te evalueren van de taal EventReactor om de van belang zijnde concepten op natuurlijke wijze te representeren.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science,

Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making*

Proof Assistants available over the Web. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14