Seamless concurrent programming of objects, aspects and events

Jurgen Michael van Ham

▶ To cite this version:

Jurgen Michael van Ham. Seamless concurrent programming of objects, aspects and events. Programming Languages [cs.PL]. Ecole des Mines de Nantes, 2015. English. NNT : 2015EMNA0118 . tel-01238752

HAL Id: tel-01238752
https://theses.hal.science/tel-01238752

Submitted on 7 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

l'université nantes angers le mans

PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

TECHNISCHE UNIVERSITÄT DARMSTADT

MINES Nantes

Thèse de Doctorat

Jurgen M. VAN HAM

Mémoire présenté en vue de l'obtention du

grade de Docteur de l'École nationale supérieure des mines de Nantes

Doktor-Ingenieur (Dr.-Ing.) de Technische Universität Darmstadt

sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27

Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)
Ecole des Mines de Nantes

Soutenue le 9 mars 2015

Thèse n° : ED 503-2015 EMNA-2015-0118

Seamless Concurrent Programming of Objects, Aspects and Events

JURY

Rapporteurs :	**M. Wolfgang DE MEUTER**, Professeur, Vrije Universiteit Brussel **M. Christophe DONY**, Professeur, Université de Montpellier-II
Examinateurs :	**M. Ulf BREFELD**, Professeur, Technische Universität Darmstadt **M. Patrick EUGSTER**, Professeur, Technische Universität Darmstadt **M. Reiner HÄHNLE**, Professeur, Technische Universität Darmstadt
Co-encadrant :	**M. Jacques NOYÉ**, Maître-Assistant, École des Mines de Nantes
Directeur de thèse :	**M. Pierre COINTE**, Professeur, École des Mines de Nantes
Co-directrice de thèse :	**Mme Mira MEZINI**, Professeur, Technische Universität Darmstadt

Seamless Concurrent Programming of Objects, Aspects and Events

Dem Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieur (Dr.-Ing.)
genehmigte
Dissertation
von

Jurgen Michael Van Ham, M.Sc.
Geboren in Bonheiden, Belgien

Doppelpromotion zwischen École des Mines de Nantes
und Technische Universität Darmstadt.

Französische nationale Thesis-Nummer: 2015EMNA0118 (http://www.theses.fr)

Referent:	Prof. Dr. Mira Mezini, Technische Universität Darmstadt
Korreferent:	Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel
Tag der Einreichung:	13.Januar 2015
Tag der mündlichen Prüfung:	9.März 2015

Erscheinungsjahr 2015
Darmstadt D17

Acknowledgements

First of all, I thank my parents for their continuing support during the hard times of these latest years.

To my remote friends spread over the globe, thank you for supporting me via many mails, even when I not always took the time to properly answer and thank you for all the support.

I would like to thank the team members from ASCOLA and AtlanMod in Nantes, for the great discussions and a social environment, which made my stay in Nantes a nice memory. My thanks also to Guilhem and Ismael with whom I shared an office and enjoyed many not always technical discussions in a great atmosphere. I would also like to thank Diana for the conversations and her support even after leaving Nantes.

My thanks to Jacques Noyé, Mira Mezini and Pierre Cointe for their guidance and for the opportunity to return for a while to the academic world. I also thank Guido Salvaneschi for his patience and all the support during all the times he helped me during my stay in Darmstadt.

Thanks to the ST group in Darmstadt, especially Andi, Edlira, Joscha, Manuel, Micha, Michael, Oliver, Sarah and Sylvia for their helpful remarks on this text.

Contents

1 Introduction **17**
- 1.1 Concurrency . . . 17
- 1.2 Language support for concurrency . . . 20
- 1.3 The design of the JEScala language . . . 23
- 1.4 Summary of the contribution . . . 24
- 1.5 Structure of the thesis . . . 25
- 1.6 Publications associated to the thesis . . . 27

I Context and Problem Statement **29**

2 State of the Art **31**
- 2.1 Aspect-Oriented Programming . . . 33
  - 2.1.1 Pointcuts . . . 34
  - 2.1.2 Advice . . . 37
- 2.2 Event-Based programming . . . 38
  - 2.2.1 Observer Pattern . . . 39
  - 2.2.2 Libraries . . . 41
  - 2.2.3 Language Integration . . . 42
  - 2.2.4 Complex Event Processing . . . 44
- 2.3 Combining Events and Aspects . . . 46
  - 2.3.1 EScala . . . 46
- 2.4 Concurrency . . . 51
  - 2.4.1 Synchronization through Shared Memory . . . 52
  - 2.4.2 Synchronization via Message Passing . . . 53
  - 2.4.3 Transactional Memory . . . 57
- 2.5 Joins . . . 58
  - 2.5.1 Calculi for concurrent programs . . . 58
    - 2.5.1.1 Pi Calculus . . . 58
    - 2.5.1.2 Join Calculus . . . 59

2.5.2 Join Languages 63
2.5.3 The role of Joins in application design 71
2.6 Conclusion 77

3 Problem Statement **79**
3.1 Case Study 80
3.2 Instrumentation 80
3.3 Coordination Logic 81
3.4 Discussion 84

II Contribution **87**

4 A Rich Event System to the Rescue **89**
4.1 Basic Concepts of JEScala 91
4.1.1 Asynchronous events with Implicit Invocation 92
4.1.2 Imperative events 92
4.1.3 Implicit events 93
4.1.4 Declarative events 94
4.1.5 Abstract events 95
4.1.6 Joins on implicit and declarative events 95
4.2 Interacting disjunctions 97
4.2.1 Disjunctions consume and fire events 99
4.2.2 Multiple disjunctions inside the same class 99
4.3 Dynamic registration of handlers 101
4.4 Summary on events 103
4.4.1 Primitive events 103
4.4.2 Sequential event expressions 103
4.4.3 Join event expressions 104
4.4.4 Inheritance 104

5 State Machines **107**
5.1 State Machines using Object-Oriented abstractions 108
5.2 State Machines using Actors 109
5.3 State Machines using events and Join Patterns 111
5.3.1 Using events 111
5.3.2 Coordination using a State Machine 112
5.3.3 Using events and Join patterns 113
5.3.4 One step further 114
5.4 A DSL to describe an FSM 115
5.4.1 FSM definition 116

5.4.2 DSL design . . . 118
5.5 Advanced State Machines . . . 119
5.5.1 Action events with arguments . . . 120
5.5.2 State events with an argument . . . 124
5.6 Conclusion . . . 125

6 Event Monitor **129**
6.1 Example . . . 132
6.2 Synchronizer . . . 134
6.2.1 Redesigning the synchronizer . . . 134
6.2.2 The synchronizer as an actor . . . 135
6.2.3 Bridging the gap . . . 136
6.3 Limitations and future extensions . . . 136

7 Implementation **139**
7.1 Event graphs . . . 140
7.1.1 Triggering an event . . . 141
7.1.2 Dynamic deployment . . . 142
7.2 Adding asynchronous event handling . . . 143
7.3 Handling disjunctions . . . 143
7.3.1 Optimizations . . . 144
7.4 FSM DSL . . . 145
7.5 Event Monitors . . . 147

III Validation and Conclusion **149**

8 Evaluation **151**
8.1 Static Evaluation . . . 152
8.2 Dynamic Evaluation of JEScala . . . 154
8.2.1 Comparison with other languages . . . 154
8.2.2 Effect of patterns complexity . . . 155
8.2.3 Effect of optimizations . . . 157
8.3 Qualitive Evaluation using a Ray Tracer . . . 158
8.3.1 Conversion of the Ray Tracer . . . 158
8.3.2 Principles of the Implementation of the Extensions . . . 160
8.3.3 Conclusion . . . 161
8.4 FSM DSL . . . 162
8.4.1 Code size . . . 162
8.4.2 Performance . . . 163
8.5 Event Monitors . . . 166

9 Related work **169**
9.1 Join languages . . . 170
9.2 Calculi . . . 172
9.3 Other languages . . . 172
9.4 Other approaches . . . 173

10 Summary and Future Work **175**
10.1 Summary . . . 175
10.2 Future work . . . 176

IV Appendix **179**

A Curriculum Vitae **181**

B Zusammenfassung **183**

C Résumé en Français **185**
C.1 Introduction . . . 186
C.1.1 Concurrence . . . 186
C.1.2 Supporter la concurrence dans un langage de programmation . . . 188
C.1.3 La conception du langage JEScala . . . 191
C.1.4 Résumé de la contribution . . . 192
C.2 L'état de l'art . . . 193
C.2.1 Programmation par aspects . . . 193
C.2.2 Programmation par événements . . . 194
C.2.3 La combinaison d'événements et d'aspects . . . 195
C.2.4 EScala . . . 195
C.2.5 La concurrence . . . 196
C.2.5.1 Joins . . . 197
C.3 La problématique . . . 199
C.4 Un système d'événements avancé à la rescousse . . . 202
C.4.1 Événements impératifs . . . 204
C.4.2 Événements implicites . . . 204
C.4.3 Événements déclaratifs . . . 204
C.4.4 Événements abstraits . . . 205
C.4.5 Join patterns avec des événements implicites et déclaratifs . . . 205
C.4.6 L'enregistrement dynamique . . . 206
C.5 Automate fini . . . 207
C.5.1 Programmation par objets . . . 207
C.5.2 Programmation par acteurs . . . 207

C.5.3 Programmation par événements . . . 207
C.5.4 Un langage dédié pour décrire un automate fini . . . 209
C.5.4.1 Automates alternatifs . . . 212
C.6 Moniteur d'événements . . . 212
C.7 Implémentation . . . 213
C.7.1 Graphe d'événements . . . 213
C.7.2 Événements synchrones et asynchrones . . . 213
C.7.3 Disjonction . . . 213
C.7.3.1 Optimisations . . . 213
C.7.4 Langage dédié . . . 214
C.7.5 Moniteur d'événements . . . 215
C.8 Validation . . . 216
C.8.1 Évaluation statique . . . 216
C.8.2 Évaluation dynamique . . . 216
C.8.3 Les automates finis . . . 216
C.8.4 Moniteurs d'événements . . . 217
C.9 Les travaux futurs . . . 218

List of Figures

1.1 Calculate the average of 3 numbers, before (a) and after (b) refactoring. 22

2.1 Examples of AspectJ method pointcuts. 35
2.2 Examples of AspectJ method pointcuts with arguments. 36
2.3 Examples of AspectJ field pointcuts. 36
2.4 Examples of AspectJ advices. 37
2.5 Simple instance of the Observer Pattern in Scala. 40
2.6 Limitation of Observer Pattern from the Java Class Library. 42
2.7 C# Delegate. 43
2.8 Memory with QT signals and slots. 44
2.9 Esper: Temperature in the capital of Greenland. 45
2.10 Esper: Maximal temperature in the capital of Greenland. 45
2.11 EventJava: Temperature in the capital of Greenland. 46
2.12 Imperative events in EScala. 47
2.13 Partial application to define a Scala function. 48
2.14 Implicit events in EScala. 48
2.15 Declarative events in EScala. 49
2.16 Only synchronous events in EScala. 50
2.17 Using synchronous events to wait. 51
2.18 Actors in Scala. 56
2.19 Future in Scala. 57
2.20 Formalization of the Join calculus. 60
2.21 Example of a Join pattern. 61
2.22 Example of competing patterns. 61
2.23 Combining Join patterns. 62
2.24 Buffer with JoCaml. 64
2.25 Buffer with Funnel. 64
2.26 Buffer with Join Java. 65
2.27 Buffer with Polyphonic C#. 66
2.28 Buffer with Joins Library. 67

2.29 Buffer and a subclass with Concurrent Basic. 68
2.30 Buffer with ScalaJoins. 68
2.31 JErlang buffer. 69
2.32 Buffer in JCThorn. 70
2.33 Languages implementing Join abstractions. 71
2.34 Lock with Joins. 72
2.35 Actor with Joins. 73
2.36 Finite State Machine (a) implemented with Joins (b). 74
2.37 Scheduler with Joins. 75
2.38 Thread-safe counter with Joins. 76

3.1 Web Server: basic components. 81
3.2 Instrumented basic components. 82
3.3 Coordination Logic with Polyphonic Scala. 83
3.4 Rate limitation as a state machine 84

4.1 The Syntax of JEScala. 92
4.2 Conflicting synchronicities for declarative events. 94
4.3 Simulation of evt e2 = !! e1. 94
4.4 Coordination Logic with Polyphonic Scala (a) and JEScala (b). 96
4.5 Distributing load among Web Applications in JEScala. 98
4.6 Additional patterns cause a deadlock. 100
4.7 Dynamic handler registration in JEScala. 102
4.8 Examples of primitive events. 103
4.9 Examples of sequential event expressions. 104
4.10 Examples of Join expressions and disjunctions. 105

5.1 On/Off switch as an FSM. 108
5.2 Sequential switch with the State pattern. 109
5.3 Concurrent switch with the State pattern and a monitor. 110
5.4 Concurrent switch implementation using actors. 111
5.5 Concurrent switch with State pattern, monitor and events. 112
5.6 Concurrent switch with events and Join patterns. 113
5.7 Repeated state machine for request-rate limiter. 114
5.8 Implementation of concurrent request-rate limiter. 115
5.9 DSL representation of a state machine. 116
5.10 DSL-based version of the state machine. 116
5.11 Adding state VeryFree to the request-rate limiter FSM. 117
5.12 Modifying the FSM by inheritance. 118
5.13 Mouse object events. 120
5.14 Naive filtering of `mousePos` events. 121

5.15 Mouse button as a state machine. 122
5.16 Filtering mouse position events to drag-only position events. 123
5.17 FSM of a reader/writer lock with at most two concurrent readers. 125
5.18 Implementation of reader/writer lock with at most two concurrent readers via DSL. 126
5.19 Implementation of a reader/writer lock without an upper bound on concurrent readers. 127

6.1 EScala example. 130
6.2 Adding concurrency to the EScala example. 131
6.3 Handlers are not more safe than other methods. 132
6.4 Handlers mutually exclusively executed. 133
6.5 Synchronizer: A practical implementation. 135

7.1 An event graph is not always a tree. 140
7.2 Deployment of event nodes. 142

8.1 Main metrics for the case studies. 153
8.2 Performance of Join languages. 155
8.3 Benchmark: Increasing complexity of matching patterns with $n = 3$ (a) and $n = 4$ (b). 156
8.4 Effect of growing pattern complexity on performance. 156
8.5 Optimization of asynchronous events in JEScala. 157
8.6 Optimization of asynchronous events with the *Thread Pool* optimization. 158
8.7 FSM for the triangle benchmark. 164
8.8 Implementation of the triangle benchmark. 164
8.9 Execution time for the triangle benchmark. 165
8.10 Sequential and concurrent universe simulation. 166

9.1 Languages implementing Join abstractions. 170

C.1 Applications web et les méthodes qui reçoivent les requêtes. 199
C.2 Création de jetons à intervalle fixe. 200
C.3 Logique de coordination. 201
C.4 Limiteur de taux mis en œuvre par un automate fini. 201
C.5 Éléments spécifiques de la syntaxe de JEScala. 203
C.6 Implémentation de CL et RL avec JEScala. 206
C.7 Automates finis de l'interrupteur on/off (a) et du limiteur de taux (b). . . 208
C.8 Interrupteur concurrent avec patron état, moniteur et événements. 208
C.9 Répetition d'automate finis RL avec JEScala. 210
C.10 L'automate fini avec notre langage dédié. 211
C.11 Ajouter l'état VeryFree au automate RL (a) et le résulte (b). 211

C.12 Temps d'exécution pour le benchmark du triangle. 217
C.13 Simulation d'univers, version séquentielle et concurrente. 218

Introduction

In this thesis, we focus on coordinating concurrency among parts of an Object-Oriented program.

1.1 Concurrency

In software, *concurrency* means that several tasks are being performed at the same time. The finest level of granularity in concurrency is made up by *threads*. Each *thread* is a sequence of instructions that is executed. While concurrency also exists among programs running in an operating system, this thesis only discusses concurrency within a user application i.e. threads. On computers with multiple processor cores, which can be part of a single processor, it is possible to execute a thread on each core. In that case, threads execute at the same physical time. However, usually the number of used threads exceeds the number of available processor cores, in that case, cores need to switch among the

execution of several threads. There are different strategies to decide on when to switch to another thread. For example, a thread can give the control to the operating system, or the operating system can take control after a short time. In our work, we rely on the latter strategy. Scheduling threads is a common situation, since current computers with less than ten processor cores often execute more than a hundred programs concurrently where each program executes at least one thread. Switching between threads requires a *context switch* [93] where a processor core needs to store the state of the current thread and load the state of the next thread to execute. The time a processing core spends on context switches cannot be used to execute programs for the user.

When several threads collaborate to perform a task, they need be to coordinated to prevent *race conditions* [86]. An example of a race condition that can occur is if two threads need to simultaneously modify a mutable collection of numbers where one thread doubles these numbers at the same time that the other thread adds two to each number. The resulting set of these operations depends on how the execution of the threads is interleaved. Unless synchronization ensures that both operations take place in the same order for all elements, the values in the collection are unpredictable.

In spite of performance penalties and the complexity of designing software that avoids race conditions, using concurrency still becomes more popular. While the original goal of concurrency was to optimize the usage of expensive hardware, using concurrency has now shifted to providing a better user experience as we explain below. However, after many years of studying and using concurrency, designing concurrent software still remains a complex task.

Originally, concurrency helped in early computers to optimize the time these expensive machines spent doing useful work, by minimizing their idle time. In the beginning, a resident monitor [96], which is the predecessor of the present-day operating system, took care of scheduling the execution of programs for multiple users. While the first programs merely processed input from punch cards, they became more interactive over time. This interactivity resulted in idle time of the computer, because interactive programs sometimes had to wait for user input before they could continue their task. As such, concurrency became a requirement to allow such idle computer time to be efficiently used by another program. Initially, concurrency only provided users a way to efficiently share expensive hardware and was not used for program interaction.

The evolution of the hardware made it possible for a user to run multiple programs concurrently. This means that unrelated programs with separate input/output data can run concurrently as well as programs modifying shared data. The latter potentially introduces race conditions. Programs at that time already interacted indirectly with their

storage and other input/output operations via the operating system. Instead of giving programs access to the physical hardware, the operating system presents the programs a model of hardware that is not shared. Programs are mostly separated from each other unless they need to interact via communication channels like pipes or network sockets. However, we cannot say that they are completely isolated, since programs that access different files on a single shared storage influence each other's progress, even if such possible delays have no effect on their final result. Programs can also share files or memory, which also makes fast communication possible. However, the programs have to correctly synchronize the access to such shared resources.

Concurrency has become an intrinsic part for a vast range of software applications. For instance, users expect that interactive graphical applications remain responsive instead of interrupting their user interaction when they perform an operation that does not finish instantaneously. Another reason for concurrency inside an application is that current multi-core hardware makes it possible to speed up a single program by dividing work into concurrently executed multiple tasks. However, the programmer has to define the interaction among the concurrently-executed tasks. Unlike the interaction among separate programs, the interaction among parts of a set of concurrent programs often circumvents the operating system, which avoids overhead. However, the operating system cannot control how these parts interact with each other. When concurrency is part of a program, the complexities associated with concurrency are no longer the exclusive problem of the operating system programmers. When collaborating cores are not only in a single computer but are spread over several computers, it is called *distribution*. In distribution, apart from concurrency, also unreliable communication and sometimes dynamic re-organization of the collaboration has to be solved. Currently, cloud computing receives a lot of interest. The interacting cores can be located in different data centers. Users and developers of applications in the cloud are not aware of changes in the group of interacting cores and their data storage. This makes it possible to optimize dynamically for properties such as price and speed. The provider of applications in the cloud could offer discounts for users that are willing to give priority to the programs of other users. When users are willing to pay a price for having their data closer to them or closer to some place where they are often accessed, it is possible to reduce communication time.

In this thesis, we focus on plain concurrency and not consider distribution.

1.2 Language support for concurrency

Synchronization can be implemented with primitives like low-level *semaphores* [29] which are explicitly managed counters to keep track on the active operations on shared resources. The slightly higher-level *monitors* [52, 58], restrict the number of active threads in a region of code, for instance an object. However, monitors are still low-level constructs that require programmers to focus on details, such as waiting to prevent deadlocks. To correctly use synchronization primitives, the programmer has to recognize each atomic group of instructions that access shared resources. This is complex and usually leads to error-prone code that is hard to understand and therefore difficult to maintain.

The history of languages with integrated support for writing concurrent user applications started with languages like Concurrent Pascal [52], which introduced monitors as part of the language. The research on concurrency has been active for approximately 40 years since then. While there are solutions for some cases of concurrency, there is no easy solution for concurrent programming in general. Therefore, implementing the correct interactions among concurrent tasks remains a difficult task. The development of new software often requires concurrency, even though it is known to be hard.

Since the correct implementation of concurrency with low-level primitives is complex, researchers have proposed language constructs that introduce a higher level of abstraction and support high-level reasoning. Examples of these constructs are Join patterns [39], Actors [56], Transactional Memory [54], Futures [7] and Data-centric synchronization [126]. In this thesis, we explore Join patterns; the other alternatives are only useful in specific situations. For example, transactional memory aims to isolate the programmer from concurrency, which has an impact on the speed of a program. The actor language Erlang [6] is successfully used in the telecommunication industry. However, the actor languages are not widely used in other fields. Futures are mostly used for concurrent computations that only interact with the thread that started them by means of a result. Data-centric programming is another recent approach for concurrent programming. Since it relies on identifying units of work that access data, new programs can be structured to benefit from this approach.

The Join Calculus [39] introduced *Join patterns* and combinations of such patterns that we call *disjunctions* as key concepts to express the interaction among a set of processes that communicate by emitting data over communication channels. The communication in the original model is asynchronous. However, synchronous communication can be built on top of its asynchronous foundation by waiting for a reply or acknowledgment

after sending data over a channel.

After the introduction of the Join Calculus, its concepts have gained high attention because they combine abstraction and practicality. They are abstract enough to overcome the limitations of low-level constructs, but are still applicable in a wide variety of scenarios. For a programmer, Join patterns are a single construct to synchronize and organize communication among the concurrent parts of software. Join patterns can express the low level synchronization constructs like locks and semaphores, while they can also express higher-level models like actors.

Because of the strengths of Join patterns, several languages have been proposed to directly support these. In the remainder of this thesis, we call these languages *Join languages*. This group of languages includes JoCaml [23], Funnel [90], Join Java [60], Polyphonic C# [9], the Join concurrency Library [108], Scala Joins [51], JCThorn [94] and JErlang [97]. The only language in this group that does not extend an existing language is the research language Funnel.

In this thesis, we argue that while the current Join languages can express the entire logic behind the coordination schemas, this logic remains interwoven with the business logic. This limitation is a result from the way that channels and emissions are being implemented by existing Join languages. These languages can implement channels in two ways, either by a method call or by a function call [23, 79, 90, 9, 60] or by using explicitly triggered events [51].

Regardless of the implementation used for a channel, interaction between the execution of the code and one or more join patterns is only possible via the explicit insertion of emission points in the correct places of the application code. Such emission points emit data via a channel to a disjunction. When methods or functions model a channel, the receiver of the channel is also hard coded by the emission point. Figure 1.1a shows as an example a Scala [91] implementation that calculates the average of three integer arguments. The code is instrumented with explicit emissions to a *divide* receiver (Lines 5, 7 and 9) and to an *add* receiver (Line 11). For this example, there are different receivers that we do not describe any further. Figure 1.1b shows the code after refactoring it to use fewer operations. We ignore the effects of rounding intermediate results. However, the emit instructions in the new code are different from the original code in Figure 1.1a. In the case the receivers are defined as part of the emission points of a channel, changing the receiver results in changes of the emission points as well. We illustrate this with the code of the same example shown in Figure 1.1a. Instead of a single observer for the divisions we need to send for each division to a different observer. For this change, the instructions on Lines 5, 7 and 9 need to be modified, each to send to its intended

```
1 def avg_first(a:Int,
2               b:Int,
3               c:Int)={
4   val a3= a/3
5   emit_divide(a)
6   val b3= b/3
7   emit_divide(b)
8   val c3= c/3
9   emit_divide(c)
10  val result= a3+b3+c3
11  emit_add(result)
12  return result
13 }
```

(a)

```
1 def avg_new(a:Int,
2             b:Int,
3             c:Int)={
4
5   val total3= a+b+c
6   emit_add(total3)
7
8
9
10  val result= total3/3
11  emit_divide(result)
12  return result
13 }
```

(b)

Figure 1.1 – Calculate the average of 3 numbers, before (a) and after (b) refactoring.

receiver.

The coordination logic in Joins-based programs is a result from the interaction via channels that connect Join patterns and data emissions points. However, because the emission points are interwoven with the rest of the code, the coordination logic is fragmented in the code as well, which implicitly creates dependencies among emissions, thus indirectly among Join patterns. Such strong dependencies prevent a modular design, but result into a monolithic program. The goals of a modular design are to help understanding and maintaining the code, and facilitating reuse. To infer logic of a coordination schema in existing programs, the programmer has to *connect the dots* by following the flow of the application and discovering how different data emissions are related and how they interact with each other. While the emissions via channels are required by Join patterns to express coordination among concurrent parts, the current Join languages support only explicit emissions that are mixed with code for the business logic. This hampers program understanding and makes programs harder to maintain. It also reduces reusabilty of parts of a program. Since the coordination logic cannot be expressed in a separate module, the changes to the coordination schemas cannot benefit from local reasoning, because the observed emissions can take place in other modules. To achieve a modular design, a solution is needed to separate the emissions from the code for the business logic. This is similar to the goal of Aspect-Oriented programming [68] (AOP) that tries to separate cross-cutting concerns into modules.

1.3 The design of the JEScala language

The language we propose in this work solves the problem of coordinating concurrent parts in a program by using the state of the art in programming languages. We exploit the synergy between Join Calculus concepts and the advanced event system of EScala [44]. The original EScala event system supports two groups of events, the *primitive* events and the declarative events. There are two kinds of *primitive* events. The first group of primitive events are the *imperative* events which are explicitly triggered by instructions in the code. The second group of primitive events are the *implicit* events, which are triggered when the execution reaches a point in the code. Implicit events are referable points in the code, which are similar to *join-points* in AOP [68]. *Declarative* events use event expressions to compose events. Examples of the supported operators include combining the runtime events and transforming or filtering them. Event expressions can contain constructs like `before(m)`, which refers to the implicit event that is triggered before executing the method `m`. Such event expressions are an alternative for a *pointcut* language in AOP. By using declarative events a programmer can select a set of multiple implicit events, similar to the use of *pointcut* languages in other AOP languages, to select pointcuts in the set of join-points that are represented by the implicit events.

Handlers are arbitrary pieces of code that are registered with an event and are implicitly invoked when the event is triggered. Events in sequential EScala are *synchronous*, because code that triggers an event waits until all direct and indirect handlers return. Indirect handlers of an event are the handlers that are registered with a different declarative event that possibly transitively use the original event as a part of the definition of the intermediate event as an event expression.

For JEScala, we extended the EScala event system to support *asynchronous* events, which do not block the sender. The Join languages that we discuss in detail in Chapter 2 support two ways of sending data via channels. The first is *asynchronous*, and the second is *synchronous*, both of which have the same semantics as in the Join languages. The channels in existing Join languages are defined by the receiving disjunction, i.e., the receiver defines the synchronicity of the channel. However, in JEScala, events represent the channels from Join languages. Since these events support dynamic registration of receivers, the receiver cannot define their synchronicity. Therefore, the synchronicity of an event is part of the definition of a primitive event, declarative events respect the synchronicity of the runtime events they combine. The varying synchronicity of declarative events is different from other Join languages where disjunctions define the synchronicity of the channels through which they receive data. By using *synchronous* as default synchronicity of a primitive event, JEScala remains compatible with EScala, which only

supports *synchronous* events.

The powerful event system of JEScala enables the modular definition of data emission sources together with their synchronization logic. This makes it possible to capture complex coordination schemas, which otherwise would be scattered in the code base. Not only does this benefit the understanding and maintainability of the program, but it also enables reuse of the same implementation of a coordination schema with different applications and vice versa.

Since the coordination schemas are expressed explicitly, it is possible for a compiler to replace the general purpose implementation by a dedicated specialized version. We explored such optimization by implementing a Domain Specific Language (DSL) that describes a finite state machine. In addition to the implementation based on events and general purpose Joins, we created versions that rely on a specialized implementation to result in faster runtime execution.

1.4 Summary of the contribution

In short, this thesis makes the following contributions:

- We motivate the need for abstractions to overcome the limitations of current Join languages that result in a scattered coordination logic, caused by the need for explicit emission sources to interact with Join patterns.

- We present the core design of JEScala, a language that exploits a seamless integration of an advanced event system with Join patterns to capture coordination schemas in an expressive and modular way.

- We study the implementation of a finite state machine (FSM). This resulted in an explicit declaration via a DSL, which makes it possible to use alternative optimized implementations. To illustrate that this DSL is limited to FSMs, we also show state machines that implement an example from functional reactive programming (FRP) and a reader-writer lock as a state machine that supports a number of readers without requiring an upper bound.

— We introduce what we call an *event monitor* that restricts the execution of a set of handlers to occur in a sequential order. The aim is similar to using object monitors for handling mutually exclusive methods in languages like Java [46] except that handlers do not block the sender of an event.

— We evaluate the design of JEScala language, our DSL for describing state machines and the use of an event monitor. We start with validating our approach by using case studies that show the validity of our design and we provide a first performance assessment. Then, we validate the DSL that we designed to describe FSMs in two ways. First, we indicate improved code metrics (less code, fewer explicitly triggering of events, less handlers); second, we use a benchmark to compare different implementations based on this DSL. Finally, we present an experiment that shows how combining parts of a sequential EScala program into a concurrent JEScala program. The resulting program run faster than its sequential ancestor from which we reused parts.

— The implementation of the JEScala language, the examples, and validation code are available on the JEScala website [63].

1.5 Structure of the thesis

Chapter 2 describes the state of the art, starting with Aspect-Oriented programming and Event-Based programming, followed by an overview on concurrent programming. Then we focus on the Join calculus. We give an overview of programming languages that use elements from the Join calculus. We do not explain the common paradigms Object-Oriented programming and Functional programming, which have been combined in the Scala [91] language. Since the main contribution of this thesis, the JEScala language, is based on Scala, we explain the elements we use, without aiming to cover the entire language.

Chapter 3 analyses the problems with the state of the art by means of a case study. The case study consists of an example that limits the rate at which a Web Server handles requests. We use this to illustrate that an implementation in a Join language requires us to modify the code of the Web Server. Such changes in existing code can easily lead to errors and also hinder futher evolution.

Chapter 4 presents the JEScala language and explains the choices that we made to integrate asynchronous events and combine them with the synchronous events that were already part of the event of EScala, a previous integration of a sequential event system into Scala.

A central part of the case study is a finite state machine (FSM). Since an FSM is a common way to express interaction, we present in Chapter 5 a DSL to describe finite state machines. Unlike the FSM used before in the case study, we also support actions as a result of a transition. When using our DSL, the description of an FSM hides the boilerplate code needed for a manual implementation. We show that this DSL can support different implementations, including implementations that are more efficient than pure Joins-based implementations. However, this DSL is restricted to events that do not carry any data. We also show the interest of FSMs built with events carrying data.

Since handlers can access shared state, we searched for a way to enforce mutual exclusion among a group of handlers. In Chapter 6, we present our approach that we call an event monitor. An event monitor is similar to an object monitor [58, 52], which groups functions or other units of code and prevents their concurrent execution. While events can occur concurrently, event monitor prevents the concurrent handling of events. Using event monitors makes it possible to reuse parts of a sequential program in a concurrent JEScala program.

In Chapter 7, we focus on the implementation. First, we explain the key elements of the implementation of JEScala. We explain how the *event graph* implements the main part of the event propagation. This graph improves performance, since it makes it possible to propagate only event occurrences that have a receiver. We also explain the three ways that we optimized the interaction between *asynchronous* event occurrences and Joins. Second, after explaining the core of JEScala, we explain how we implemented the DSL for finite state machines and the different types of code this DSL generates. Third, we explain an implementation of the event monitor.

Chapter 8 is devoted to the evaluation of the proposed language abstractions. We show how programs implemented with JEScala exhibit improved design metrics compared to programs using other Join languages. This better design results in programs that are easier to understand and maintain. We represent Join languages with a subset of JEScala, which makes it possible to focus on the differences that result from the powerful event system that JEScala adds to Join languages. We also compare the performance of JEScala with other languages that do not support implicit invocation.

Next, we evaluate our DSL to describe a finite state machine. To indicate how many lines our DSL saves in describing a FSM compared to a full Joins-based implementation, we derive formulas to compare the code size of a Joins-based finite state machine with the size when using our DSL description. Another motivation to use a DSL to explicitly describe an FSM is that the domain knowledge about an FSM makes optimized implementations possible. We look at the effect of using various ways of programming a synthetic FSM, including using our DSL and its different implementations. We conclude our evaluation by showing how we can reuse parts of an existing sequential EScala program to build a concurrent program that executes faster.

In Chapter 9, we discuss the related work on approaches to coordinate concurrent parts in a program. This includes the Join languages that we discuss in Section 2.5, but also a language, MogeMoge [87], for a game engine that uses a single global disjunction. We also mention the language Pāñini [75], which is not based on Joins, but aims for implicit concurrency. Approaches like Implicit invocation with traits [95] and Join point interfaces [15] use events but do not explicitly mention concurrency. Constructs like Sequential Object Monitors [21] and Parallel Object Monitors [20] are an easier to use alternative to classic object monitors as provided by Java.

Finally, Chapter 10 summarizes the thesis and outlines areas of future work. We split the future work into different groups. First, there is theoretical work to define a model for this language; second, we suggest possible optimizations; third, a modified the compiler makes the language available to other users; fourth, the exploration of constructs like windows over time or over a number of events inspired by Complex Event Processing (CEP) [77]. We also suggest to explore extending the event system with return values. Finally, we suggest to further explore the application of the language.

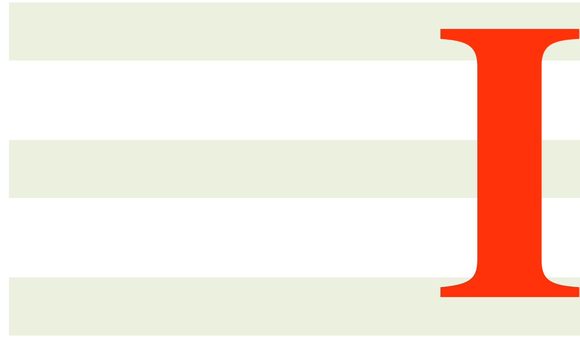
1.6 Publications associated to the thesis

An early integration of Join patterns into the EScala event system was submitted to the AOSD '12 Student Research Competition [124], leading to a Third Place award.

Reconsidering the initial ideas about Joins led to the language described in Chapter 4, which was presented at Modularity '14 [125].

The study of finite state machines in Chapter 5 builds further on a still unpublished paper.

The event monitor is based on work in progress to adapt the Actor model [56] to support concurrency within the EScala event system.



Context and Problem Statement

State of the Art

Contents

2.1 Aspect-Oriented Programming **33**
2.1.1 Pointcuts 34
2.1.2 Advice 37
2.2 Event-Based programming **38**
2.2.1 Observer Pattern 39
2.2.2 Libraries 41
2.2.3 Language Integration 42
2.2.4 Complex Event Processing 44
2.3 Combining Events and Aspects **46**
2.3.1 EScala 46
2.4 Concurrency **51**
2.4.1 Synchronization through Shared Memory 52
2.4.2 Synchronization via Message Passing 53
2.4.3 Transactional Memory 57
2.5 Joins **58**
2.5.1 Calculi for concurrent programs 58
2.5.2 Join Languages 63
2.5.3 The role of Joins in application design 71

2.6 Conclusion 77

While Object-Oriented programming [105, 25] (OOP) helps developing modular software, OOP has its limitations. To overcome these limitations combinations with other paradigms like Aspect-Oriented programming and Event-Based Programming were studied.

Some concerns in classic Object-Oriented programs are inextricably woven in the code. Aspect-Oriented Programming [68] (AOP) aims to unravel these concerns into aspects. AspectJ [67] was the first language designed to support Aspect-Oriented Programming. We use AspectJ to show the concepts of AOP that we use in the language EScala below.

Event-Based programming facilitates modular software by weakening the coupling between senders of information and its receivers or, in other words, between subjects and receivers of that information. The central concept in event-based programming is implicit invocation [43], which means that a subject broadcasts information that any interested observer can receive.

We provide an overview of the language EScala [44], which integrates advanced events from Event-Based programming and Aspect-Oriented programming into the Object-Oriented Scala [91]. The advanced event system of this language supports events that result into other events. For example, two distinct events (e.g. rain or thunder) can result into a new single event (e.g. bad weather).

We give an overview of the classic concurrency models, before we explain high-level concurrency that is based on the Join calculus [39]. The Join calculus is a formalism for concurrent calculations that was created to support the notion of locality in distributed programs. Its asynchronous events are not only fit for communication with other hosts, we use them for concurrency without distribution. Asynchronous events and detecting patterns of events are the elements of the Join calculus that are integrated into programming languages that we call Join languages. We give an overview of the Join languages. We show examples based on Joins to illustrate concurrency based on the Join calculus. Joins express synchronization patterns in a declarative way. Joins provide an alternative for the low-level synchronization based on semaphores and explicitly managed threads.

This chapter is organized as follows. Section 2.1 presents Aspect-Oriented Programming. Section 2.2 presents event systems. Then we show the classic approaches to concurrency in Section 2.4. Section 2.5 introduces the Join calculus and languages based on it.

2.1 Aspect-Oriented Programming

Classic Object-Oriented approaches cannot implement all concerns as separate modules. For example, a separate module cannot log method calls. To add logging to methods of an object, its class needs to be modified. For example, logging the name of each executed method of an object is possible by inserting a call to some logging component at the start of each method. This logging code is *scattered*, since it is part of all methods. As a result, the programmer needs to add logging to every additional method to make sure logging works correctly after the change. The code for one concern is targeted writing code for other concerns.

Kiczales et al. [68] call this kind of concerns *cross cutting* concerns. The code responsible for their implementation is *scattered* throughout the code for other functions. As a result of combining different tasks, the code becomes *tangled*. The same issue appears again if a second concern, for instance concurrency is introduced. Aspect-Oriented Programming (AOP) was introduced as a way to isolate such concerns in separate modules.

The following metaphor explains Aspect-Oriented Programming and a part of its terminology. Weaving fabric usually starts by combining (weaving) equal threads into fabric, languages like HyperJ[92] use a symmetric view and weave equal threads (aspects) resulting into fabric (program). However, the asymmetric view is more common, where aspects are woven into the existing fabric of a *base* program.

In the early years of Aspect-Oriented Programming, Filman et al. [38, 22] defined Aspect-Oriented programming as the combination of *quantification* and *obliviousness*. The word *obliviousness* means that the programmer can write a base program without worrying about aspects that are woven into it. However, later not all authors [104] agree with this. Steinmann [113] explains that there is no strict modularity, since changes in either the base program or in the aspects can break the program.

Weaving can take place at different moments. AOP can be purely static (i.e. take place at compile time), for instance AspectC++ [111] that uses a preprocessor to combine the source of a base program with aspects and weave them into C++ source files. This approach makes it possible to use unmodified existing compilers. AspectJ [67] modifies the byte code when loading classes to weave the aspects into the base code. The other extreme is to delay weaving until class loading or JIT compilation [98] at runtime, this supports the flexibility for dynamic weaving. The latter requires support from the virtual machine. PROSE [99] uses the debug interface from the virtual machine for runtime weaving, a modified virtual machine Steamloom [13] offers a more efficient alternative.

Meta Object Protocols [66] (MOPs) are a predecessor of AOP. Meta Object Protocols give the programmer control over the interpretation of a program. Language implementations with an MOP give the programmer control over the interpretation of a program

using *meta-objects*. MOPs can be used to implement a form of dynamic AOP but without the linguistic support provided by AOP [118].

AspectJ [67] was the first AOP implementation for Java and is widely used. AspectJ is the de facto standard for AOP. Therefore, we explain its *pointcut* and *advice* model [81, 80]. The contribution of this thesis is based on the language EScala, which contains elements from AOP. To explain the AOP part of EScala we refer to the concepts of AspectJ, which we explain below.

Another early implementation for AOP in Java is AspectWerkz [16]. Initially this was different from AspectJ. However, both systems shared the same ideas and grew close to each other. In the year 2005, these languages merged into AspectJ 5. Before this merge, both languages supported next to their own way of describing pointcuts, aspects and advices an annotation based description.

The initial version of AspectWerkz uses XML-files to describe aspects, pointcuts and advices. The frameworks JBoss AOP [62] and Spring [127] also use XML-files.

There are alternative languages that offer additional features. For instance, CaesarJ [4] supports dynamic deployment. AOP does not always build on top of Object-Oriented Programming languages. There are languages that are not Object-Oriented, but their programs still can act as a base program, for example Cobol [70] and ML [26].

Aspects specify places (pointcuts [81]) to insert a code snippet (advice [81]). We illustrate below both concepts with AspectJ, which uses Java [46] as its base language. An aspect is similar to a class. However, in addition to encapsulating variables and methods, it also encapsulates pointcuts and advice. The pointcuts in AspectJ refer to a piece of code and not to a place between pieces of code. Because of this there are different types of advice that we explain after showing the main pointcuts.

After introducing event systems (Section 2.2), the execution of program triggers events, that are implicitly defined as part of that program. By observing and reacting to those events additional code can be executed during runtime. Using events for AOP has been studied before by different authors [32, 128, 103].

2.1.1 Pointcuts

In AOP, the term *join-points* refers to a point during the execution of a program. The set of join points of a program is determined by the language in which that program is written.

Pointcuts are query-like expressions that select a set of *join-points* based on their properties. Different pointcut languages are compared by Stoerzer et al. [115]. For example,

the pointcut language in AspectJ is a combination of patterns for signature of methods and expressions of predicates over the set of *join-points*. We use AspectJ for the examples below.

A complete overview of the *kinds* of poincuts that are supported by AspectJ can be found in [119]. The *primitive* pointcuts are predicates, that can be composed using logical operators. The primitive pointcuts include the following:

— Method invocation and execution

— Field accesses

— State

— Exception Handling

— Initialization, pre-initialization and static initialization.

— Control Flow

Not all these kinds of pointcuts are used in the rest of the work. Therefore, we focus on the pointcuts related to methods and state.

```
1 pointcut publicIntCall():
2       call(public * *(int)) ;
3
4 pointcut publicIntExec() :
5       execution (public * mth*(int)) ;
```

Figure 2.1 – Examples of AspectJ method pointcuts.

Pointcuts related to methods Figure 2.1 shows definitions of pointcuts related to method calls. A method call is a relation between caller and a called method. To observe a method call, AspectJ can observe both sides of this relation. Pointcut definitions start with the keyword `pointcut` followed by the name of the pointcut and between parentheses the arguments. Each of the two pointcuts in Figure 2.1 selects calls to public method with a single `int` argument. The first pointcut, `publicIntCall` (Line 1) selects calls to such a public method. While the second pointcut (Line 4) refers the receiver side that executes such a method. Both pointcuts use a pattern to specify method(s) they observe. This pattern specifies the signature of a method, additional flexibility can be added by replacing elements in the signature by an asterisk as wildcard. The wildcard can also replace a part of an element, for example to specify a prefix or postfix. The pattern for the `execution` pointcut shows how to restrict this selection to only include methods with a name prefixed with the letters `mth`. Instead of the asterisk between `public` and the method name, a pattern can define a specific return type.

```
1 pointcut publicIntCall(int i):
2       call(public * *(int)) && args(i);
3
4 pointcut publicIntExec(int i) :
5       execution (public * mth*(int)) && args(i) ;
```

Figure 2.2 – Examples of AspectJ method pointcuts with arguments.

Pointcuts related to state Primitive pointcuts can intercept methods at caller or the receiver side. However, these pointcuts cannot extract information like calling object (`this`), the receiving object (`target`) and the arguments of the method call. In an Object-Oriented language, both objects can depend on choices made during runtime. In AspectJ, three pointcuts extract information related to a method call. A pointcut can select the calling (`this`) object or the receiving `target` object. The same is possible for the arguments of the method call by using the `args` pointcut.

These three pointcuts are part of an expression. Figure 2.2 shows two expressions (Line 1, 4) that combine the primitive pointcuts from Figure 2.1 with the `arg` pointcut. The expression at the right-hand side of the column specifies the join points. This expression combines predicates that are separated with a && or ||. It composes one or more pointcuts. Next to the and (&&) and or (||) operators exists a not (!) operator. These operators combine predicates into a boolean expression, that is used as a right-hand side in a pointcut definition.

The examples in Figure 2.2 extract the argument from each method call as part of the expressions that define the pointcuts and provide that extracted information as an argument of the pointcut.

```
1 pointcut setXPnt(int):
2       set(int Point.x) && args(x)
3
4 pointcut getXPnt():
5       get()
```

Figure 2.3 – Examples of AspectJ field pointcuts.

Pointcuts related to fields The two pointcuts in Figure 2.3 refer to the accesses to the variable `x` of the instances of the class `Point`. The pointcut `getXPnt` (Line 4) refers to reading variable `x`. The pointcut `setXPpnt` (Line 1) refers writing that `x`. The two

pointcuts from Figure 2.1 do not specify a class, unlike the examples in Figure 2.3, that observe accesses to a variable. The latter are more precise by defining the class, which is not always required.

In the pointcut of Figure 2.3 the argument of the pointcut is retrieved by the `arg(x)` pointcut. While for the last pointcut (Line 4 in Figure 2.3) there is no argument, hence it cannot be extracted.

2.1.2 Advice

Pointcuts [81] define places to insert code, which is provided by an advice. An advice is a piece of code to insert relative to a pointcut. We show below examples of the three *kinds* of advice, which correspond to different ways of altering the execution of a joinpoint. A `before` advice is executed *before* join points selected by a the pointcut. An `after` advice *after* and an `around` advice *around* (which as we well see, requires some further explanations).

```
1 before(int x) : publicIntExecs(x) {
2     System.out.println("method executed "+x);
3 }
4
5 after(int x) : setXPnt(x) {
6   if (x > 10) throw new IllegalArgumentException("X too big");
7 }
8
9 around(int x):setXPnt(x) {
10   proceed(x*2)
11 }
```

Figure 2.4 – Examples of AspectJ advices.

Figure 2.4 shows how to specify an advice that inserts code at a pointcut. An alternative is to define the pointcut in place instead of its name. A before or after advice starts with respectively the keyword `before` or `after`, optional arguments between parenthesis to extract parameters from a pointcut. After a column follows either the name of a pointcut, or the definition of a pointcut. Finally, there is the piece of code to insert. This piece of code can use the arguments that are extracted from the pointcut. Line 1 adds logging before the execution of the pointcut `publicIntExecs` that we defined in Figure 2.1 as a call to a method with a single `int` argument.

Since it is possible to insert code before and after a pointcut, it is possible to combine these in wrapping code `around` an advice. Line 9 in Figure 2.4 shows this kind of advice. The `proceed` replaces in the advice the code instruction that was selected by the pointcut. We use the `setXPnt` from Figure 2.3 that points to the method `set`. The pointcut `setXPnt(x)` replaces the method `set` by the body of the advice, this body calls the original method `set` with the transformed argument `x*2`.

2.2 Event-Based programming

In Event-Based Programming, the program flow is driven by reacting to events. Events in this case denote noteworthy changes in the state of the program. The code where events take place does not make any assumption on the presence nor on the number of reactions that the events cause. An observer can express its interest for certain events, after which these events invoke reactions that are defined by this observer.

For instance, an event in this paradigm can be a variable that changes value or note specifically that takes a given value. In Graphical User Interfaces (GUIs), events can be mouse clicks or changes of the mouse coordinates due to the entering or leaving a certain part of the screen, which changes the mouse pointer. Events can be exceptions like a division by zero or the impossibility to allocate memory.

Programmers define *handlers* to implement a desired reaction for an event. After registering a handler with an event, the handler executes when the event occurs. In other words, the handler is executed because the event was *triggered*. In Object-Oriented Programming, a handler is usually a method. However, languages like Scala use functions to implement handlers.

Implicit invocation Events *implicitly* invoke handlers. With method invocation, the caller requires knowledge about the method target (callee). With implicit invocation, code can trigger events without making any assumption on the number of reactions. This means there can be zero or more reactions for each event. This flexibility makes events useful in a modular design where the receivers are not yet known during the design of a module. This was studied before by Garlan and Notkin[43].

Another view on implicit invocation is that an event results in publishing a notification and that observers react to these notifications after they subscribed to them. Eugster et al. [36] studied publish/subscribe systems and propose two classifications. Their first classification mentions three ways of decoupling between publisher and subscriber: synchronization, location and time. By decoupling synchronization the publisher does not wait for any feedback from the subscriber. Location decoupling is what we call

distribution, i.e., the participants are spread over more than a single host. Time decoupling means that publisher and subscriber do not need to be active at the same time, the subscriber can receive messages that were sent before it became active. The second classification described by Eugster et al. identifies three ways used by subscribers to express their interest in events. The first used is topic-based. A topic is a communication channel to which a subscriber can tune to receive its messages. Content-based subscription uses one of more attributes in the message. The third system they describe is type-based, kinds of event are associated with a type, which results in a tighter integration into a programming language.

An early example of publish/subscribe in Object-Oriented Programming is the Model-View-Controller [69] (MVC) pattern. MVC was introduced into the Smalltalk [45] class library to support the creation of Graphical User Interfaces (GUIs). The MVC pattern uses three groups of classes: models, views and controllers. The relations among instances of these classes are dynamic. For example, a user can create different views on a model that represents a three dimensional object in a Computer Aided Design (CAD) program. The instances from each of these groups publish events to which the objects of another group can register and unregister to adapt to changing configurations.

In Sections 2.2.1–2.2.3, we discuss topic-based publish/subscribe in systems without any decoupling on synchronization, locality or time. In Object-Oriented Programming they are often implemented by the Observer [42] pattern. Then we describe in Section 2.2.4 Complex Event Processing [77] (CEP) systems that decouple synchronicity, time and location, these systems use type-based subscription. We discuss CEP since this supports transforming and filtering events and detecting patterns of events. CEP systems detect patterns in events streams like sensor readings or trade information from stock exchanges. In Section 2.3.1, we describe EScala, a language that integrates the combination of AOP-style implicit events with concepts of a CEP system into an Object-Oriented language. This language is used as a base for JEScala that is part of the contribution.

2.2.1 Observer Pattern

The Observer pattern [42] is a standard way to implement a *one to many* relationship among objects in an Object-Oriented language. In this relationship, the changes in a single object, that has the *subject* role, notify the objects that fulfill the role of *observer*. These notifications are sometimes called events. This pattern consists of two classes, one for each role.

Figure 2.5 shows an example of using the Observer pattern in Scala. We use this implementation to introduce Scala to the reader. Classes in Scala start with the keyword

```
class Subject {
    val observers=new Set[Observer]()
    def registerObserver(obs:Observer)= {
        observers += obs
    }
    def publish(msg:Msg)= {
        observers.foreach(obs => obs.update(msg))
    }
}
class Observer {
    def update(msg:Msg)= {
        ..
    }
}
object Program {
    def run()= {
        val s=new Subject
        val obs=new Observer
        s.registerObserver(obs)
        s.publish(new Msg("Does anybody hear me?"))
    }
    def main(args:Array[String])= {
        run()
    }
}
```

Figure 2.5 – Simple instance of the Observer Pattern in Scala.

`class`, which is similar to Java. Scala supports variables and values. The variables are similar to Java variables. The immutable values are declared with the keyword `val`. Line 2 initializes the value `observers` with a new instance of `Set`, using `Observer` as type parameter. While the value `observers` is immutable, the content of the empty set can change. Similar to C++ variables declared with the `auto` type, the compiler infers the type for the value `observers`. Line 3 declares a method by using the keyword `def`, the formal parameters between parentheses are argument names followed by a colon and the type. After the sequence of arguments optionally follows the return type after a colon. In most cases the return type can be inferred, unless it is a recursive method. Then follows an equal sign after which the method body follows. Unlike in Java the curly braces around a body with single instruction are not mandatory. A subject exposes the `publish` method (Line 6), and the `registerObserver` method (Line 3) to

maintain its internal collection of Observer instances. For simplicity, this implementation can only register observers, unregistering observers is omitted. The Observer class represents the second part of the pattern, it exposes an update method (Line 11), which is called by the publish method of an observed Subject instance.

In later examples, we indicate the entry method with the name run (Line 16). However, this is only to reduce the size and the complexity for the reader. Similar to a Java program, the execution of a user program in Scala starts with the main method (Line 22). The main method (Line 22) has to be part of a singleton instance. Other examples use the run method, therefore we do not use its body for the main method, but call the run method instead. In Scala, this is created by using the keyword object (Line 15) instead of class. Java static members are in Scala part of a singleton object. This singleton object is called a *companion* object of the Scala class with the same name.

Because the Observer pattern is a way to implement implicit invocation, it can be a building block for an event system. In this system a subject can represent an event. This makes it possible to add reactions implemented in observers without modifying the code that contains the subjects.

2.2.2 Libraries

Libraries can reduce the amount of code bloat for the Observer pattern. For example, both the Java Class Library or the .NET Framework support this pattern. The Java Class Library offers an interface Observer and a class Observable in the java.util package. However, when a single observer observes two subjects or more, additional code is required to handle the updates from different subjects. The example in Figure 2.6 is a raft (Line 1) that floats on a two dimensional pool with a Sensor (Line 9) at both sides. The update method (Line 2) of the raft needs to recover the sensor. Additional sensor instances increase the complexity of this method.

The Swing library uses specialized implementations instead of the generic version from the Java Class Library. These are *listeners* that react to events defined inside the Swing library. The programmer uses predefined interfaces to instantiate anonymous classes. The subjects are in this case part of the library, extending this approach for other events involves additional boilerplate code, similar to what the class library provides for the existing Swing events, including interfaces for its listeners.

```
1  class Raft extends java.util.Observer {
2     void update(Observable o, Object arg) {
3        if (..)    goToLeft(arg)
4         else       goToRight(arg)
5     }
6     void goToLeft(object arg) { .. }
7     void goToRight(object arg) { .. }
8  }
9  class Sensor implements java.util.Observable {
10    public Sensor(Raft r) {
11       addObserver(r);
12    }
13
14 }
15 public void run() {
16    raft=new Raft();
17    Sensor left=new Sensor(raft);
18    Sensor right=new Sensor(raft);
19    left.notifyObservers();
20 }
```

Figure 2.6 – Limitation of Observer Pattern from the Java Class Library.

2.2.3 Language Integration

As we showed above, using the Observer pattern as part of a library still requires the programmer to write boilerplate code. To reduce the complexity of this additional code for the programmer, some programming languages contain support for implicit invocation, such as the delegate in C# [53] or the signals and slots in Qt [28] framework.

C# Figure 2.7 shows at Line 2 a `delegate` that defines a return type and arguments, similar to a method signature. Like a class, a delegate has to be instantiated before it can be used. In C# first class constructs start with a capital. Instantiating a delegate (Line 6) assigns a variable of the type defined by the declaration of the delegate, at the right hand side of the assignment is the name of the delegate and between parenthesis a method that has compatible types. The syntax looks similar to instantiating a class. However, it is possible to add (Line 7) additional receivers to the same delegate by using the += operator.

```
1 class DelegateSample {
2    public delegate void Subscriber(int I);
3    void receiver1(int I) { .. }
4    void receiver2(int I) { .. }
5    public void run() {
6       Subscriber Publish=new Subscriber(receiver);
7       Publish += new Subscriber(receive2);
8       Publish(2);
9    }
10 }
```

Figure 2.7 – C# Delegate.

Invoking an instance of a delegate (Line 8) looks similar to calling a method, however, there is no receiver defined. When using delegates, only the type of the method is important, which avoids the boilerplate code with interfaces in the observer pattern.

Qt Trolltech developed the initial version of the portable Qt [28] framework for graphical user interfaces. This framework uses C++ classes with additional `signals` (events) and `slots` (handlers), which are additional members for Qt classes. The programmer registers slots with signals by using `connect` statements. Figure 2.8 implements a memory cell. The class `MemoryCell` inherits from `QObject` (Line 2), which is the Qt base class. Instances store their value in the variable `m_value` (Line 19). It has a getter (Line 7) and setter (Line 8) method. The `setValue` method is not only a setter, it also emits a signal using the old value (Line 9) as it argument. This signal `valueChanged` is defined at Line 17. The memory cell has a slot `changeValue` (Qt terminology for what we call a handler) that set the value to the received value. Line 22 creates two `MemoryCell` instances as the automatic variables `mc1` and `mc2`. The slot `changeValue` of `mc2` is connected to the signal `valueChanged` of `mc1` at Line 23. When we change the value of `mc1` (Line 24), `mc2` is notified of its own value. By connecting additional `MemoryCell` objects, the resulting chain of objects can remember extra steps.

The Meta Object Compiler (MOC) transforms Qt source files with these extended classes to standard C++ source files. This transformation relies the `Q_Object` macro (Line 4) and implements the slot and signal members in normal C++ code. The normal tool chain for C++ can use this preprocessed code.

```
1  #include <QObject>
2  class MemoryCell : public QObject
3  {
4      Q_OBJECT
5  public:
6      MemoryCell() { m_value = 0; }
7      int value() const { return m_value; }
8      void setValue(int n) {
9        emit valueChanged(m_value);
10       m_value=n;
11     }
12 public slots:
13     void changeValue(int value) {
14       setValue(value);
15     }
16 signals:
17     void valueChanged(int newValue);
18 private:
19     int m_value;
20 };
21 // usage
22 MemoryCell mc1,mc2;
23 connect(&mc1,SIGNAL(valueChanged(int)),&mc2,SLOT(changeValue(int)));
24 mc1.setValue(2);
```

Figure 2.8 – Memory with QT signals and slots.

2.2.4 Complex Event Processing

It is possible to transform and filter the arguments of events before reacting to the resulting event. Patterns of events result into new events. Complex Event Processing [77] (CEP) uses these operations to extract high-level events from low-level events. As an example, we consider a set of points that are spread over the area of a city or even larger. At each of these points, a barometer measures the air pressure and emits the read value. Barometers need calibration, which can be implemented by transforming the values by using the values that are collected on a calm day. However, sometimes they break, then they only emit 0 values, this can be solved by filtering these values. After these operations, we can derive the direction and speed of the wind. Combining wind information with rainfall of those same points, can result in short time rain predictions for the observed area. The individual air pressure measurements are in this case low-level events,

while the high-level events are the wind speed and direction, which are derived from these low-level events.

Several Complex Event processors like SASE [129], Cayuga [27] and Esper [35] exist. Similar to programs that access a database via SQL-statements, programs use a query language to interact with these CEP processors. Unlike for databases, each implementation has its own SQL-like query languages. The queries in these languages use event streams instead of database tables.

Esper [35] is an open source implementation for CEP that is used by enterprises. Since the three mentioned system use a similar language, we do not need to show examples for each of these CEP systems. Because Esper can be downloaded, we choose Esper for the examples below. Like the other CEP engines we mentioned, Esper uses a query language *EPL* (Event Processing Language) that is inspired by SQL, the other systems use other names for their variation of this language.

```
1 insert into greenlandTemperature
2   select (celsius-32)*5/9
3     from celciusSensor
4     where location="Nuuk"
```

Figure 2.9 – Esper: Temperature in the capital of Greenland.

Figure 2.9 shows as an example that extracts the temperature in Fahrenheit from Nuuk, the capital of Greenland. The input is a stream with measurements in degrees Celsius (centigrades) and locations. The results are made available via the stream `greenlandTemperature`, by removing the `insert` clause (Line 1). A programmer extracts values from a CEP system in a similar way as he retrieves value from a database.

```
1 select max((celsius-32)*5/9)
2   from celciusSensor:win(1 day)
3   where location="Nuuk"
```

Figure 2.10 – Esper: Maximal temperature in the capital of Greenland.

A combination of low-level events can result into high-level events, optionally these events need to occur within an interval of time. Esper can define windows on an event stream, these windows can be expressed as time or by the number of events. The events during a time interval can be retrieved from such windows. Figure 2.10 shows how to observe a stream via a time window of a day. The example relies on this feature to extract the maximum temperature during the last day.

```
1 class Thermal {
2   event celciusSensor(String location, Double celsius)
3     when (location=="Nuuk") {
4         greenLandTemperature((celsius-32)*5/9);
5     }
6 }
```

Figure 2.11 – EventJava: Temperature in the capital of Greenland.

EventJava [37] integrates distributed CEP into the programming language Java. Figure 2.11 shows how a new event `greenLandTemperature` is triggered like a method with the argument that is converted from Celsius to Fahrenheit, when the location of the received event is the name of the capital of Greenland. The keyword `event` defines an event, which can be triggered like a method. The keyword `when` filters events and can execute code, which in this case triggers the assumed event `greenLandTemperature`.

An alternative approach consists of embedding these facilities into a complete language. This approach was followed by the project that initiated CEP: Rapide [120]. Unlike SASE, Cayuga and Esper, it is not an engine similar to a database, instead it is a complete programming language to build the whole application.

2.3 Combining Events and Aspects

Aspect-Oriented Programming and Event-Based Programming are related, since they both can decide after writing the code for a program, to execute additional code at runtime. However, they use a different terminology for similar concepts. Steimann et al. [114] mention the equivalences between for example an *event* and a *joinpoint* or an event *handler* (reaction) and an *advice*. This relation is used in programming languages like Ptolemy [103], JasCo [116], EventCJ [65] to express joinpoints as events. An advice in these languages is implemented by associating a handler to the event that represents the desired pointcut. However, these languages use a simple event system. This means that handlers can be associated with events, but there is no declarative way to filter or transform events before handling them.

2.3.1 EScala

The programming language EScala supports AOP via events like the languages we mentioned before. However, EScala supports advanced events via its event expressions. It

integrates an advanced event system into an Object-Oriented language. Events in this language are members of an object. Since EScala is the language that is used as the foundation for the language developed in this thesis, we explain it here via examples.

The word *event* in EScala refers to a *source construct*, similar to what CEP systems like Esper [35] call an event source. Triggering an EScala *event* results into a runtime event *occurrence*. When it is obvious that we refer to the runtime event *occurrence*, we abbreviate this to the word *event*.

```
1  class Imperative {
2     imperative evt publish[Int]
3     private def receiver(arg:Int):Unit={
4        println("received␣:"+arg)
5     }
6     publish += receiver _
7     def run()={
8        publish(10)
9     }
10 }
```

Figure 2.12 – Imperative events in EScala.

The *primitive* events in EScala are similar to the events produced by Observer pattern or C# delegates. There are two groups of primitive events. The first group, the *imperative* events are explicitly triggered by the program. The event `publish` in Figure 2.12 (Line 2) is declared with the type `Int` for its argument. The observers of events in EScala are functions with an argument type that is compatible with the parameter of the event they observer. These functions do not return a value. The underlying Scala represents this by the return type `Unit`. In Scala, the return type `Unit` is similar to the Java return type `void`. The `receiver` method (Line 3) provides the code for a handler, which prints its argument. However, the handlers in EScala are not methods but functions. The underlying Scala provides *partial application*.

Figure 2.13 shows in the `run` method *partial application* and its use in Scala. Line 5 calls a method `meth` of object `o` with the value `2` as argument. Using Scala functions as handlers has the advantage that these do not require a receiving object. Line 6 defines a Scala function `fun1` that includes the receiving object `o`. Partial application on Line 7 shows a shorter form to define the Scala function `fun2`, that does the same as `fun1`. Finally, Line 8 shows how to use the function `fun2`, that we defined at Line 7.

Handlers use the += operator to register with an event. Figure 2.14 shows on Line 5 the registration of the method `receiver` as a handler with the event `publish`, which is

```
class aClass {
  def meth(a:Int):Unit={..}
}
def run(o:Aclass)={
   val res1=o.meth(2)
   val fun1=((arg:Int) => {o.meth(arg)} )
   val fun2=o.meth _
   val res2=fun2(2)
}
```

Figure 2.13 – Partial application to define a Scala function.

topic-based subscription for classification of publish/subscribe [36]. For the programmer the underscore after the name of the method is important, since it triggers partial application of a method to create a function. The method `run` (Line 7) shows that triggering the event with an integer value is for a programmer similar to calling a method with an argument of the type of the event.

```
class Implicit {
  observable def move(target:Point)= {..}
  evt moved = before(move)
  def tracer(p:Point):Unit = println("now at "+p)
  moved += tracer _
  def run()={
    move(Point(3,2))
    move(Point(2,1))
  }
}
```

Figure 2.14 – Implicit events in EScala.

The second group of primitive events are the *implicit* events. The execution of a program triggers the *implicit* events when its flow reaches such an event. These implicit events are part of the program similar to the join points in Aspect-Oriented Programming. Figure 2.14 shows the method `move` (Line 2), which is marked as `observable`, this inserts events before and after the execution of the method. The event `moved` (Line 3) is defined as an alias for the implicit event that is inserted before the execution of the method `move`. The method `tracer` shows that handlers receive the arguments of the method call, in case they observe the implicit event before that method. The implicit event that is triggered after executing a method has a parameter of type `Pair`. This `Pair`

contains the arguments and the return value of that method call. Similar to imperative events, handlers like `tracer` can register with implicit events by using the += operator (Line 5).

Registered handlers can unregister with an event, via the −= operator in similar way as the += operator registered the handler.

```
1 class Declarative {
2   imperative evt source[Int]
3   imperative evt altsource[Double]
4   evt odd=source && ((arg:Int) => (arg%2)==1)
5   evt half=source map ((arg:Int) => arg/2)
6   evt union=source || altsource
7   def run()={
8     source(10)
9     altSource(4.2)
10  }
11 }
```

Figure 2.15 – Declarative events in EScala.

Apart from the primitive events, EScala supports declarative event expressions, which add some features from Complex Event Processing (CEP) to EScala. These expressions can combine, transform and filter event occurrences. We illustrate these operations by examples in Figure 2.15.

The declarative `odd` event (Line 4) filters the event occurrences from the event `source` (Line 2) using the operator && followed by a predicate function. Its argument is compatible with the argument of the `source` event and returns `true` or `false`. The event `odd` emits only the odd values that triggered the `source` event.

Declarative events can transform event occurrences by using the `map` operator. The declarative event `half` (Line 5) transforms the event occurrences from the event `source` by applying the function that follows the `map` operator on the parameter of each event occurrence. The argument of this function is compatible with the argument of the event `source`. The return value of this function defines the parameter type of the resulting event `half`. In our example, the event `half` emits the values that triggered the event `source` divided by two.

The EScala event expressions can combine the event occurrences from two events by using the binary union operator (||). The example in Figure 2.15 defines the event `union` (Line 6), it combines the occurrences from the events `source` and the event

altsource. The type of the argument of the event union is compatible with the arguments of both events (source en altsource). In our example the handler needs to deal with arguments of the type Double. We note that a resulting type Any can accept any type of argument.

```
1 class Synchronous {
2   evt experiment[Unit]
3   def sleepMethod()={
4     Thread.sleep(1000)    // 1 second
5   }
6   def sleepHandler():Unit= {
7      sleepMethod()
8   }
9   experiment += sleepHandler _
10  def run()= {
11    sleepMethod()
12    experiment()
13  }
14 }
```

Figure 2.16 – Only synchronous events in EScala.

When we apply the first of the two classifications described by Eugster et al. [36] for publish/subscribe systems on the EScala event system, it does not decouple a publisher from its subscribers for any of the three types of decoupling in the classification: synchronization, location nor time. The example in Figure 2.16 shows that synchronous events and method calls wait until their handlers return. When the method sleepMethod (Line 3) is called, the program only continues after that this method returns. Similarly, when triggering a synchronous event, the program only continues after all registered handlers of the synchronous event returned. Figure 2.17 shows a naive example how the synchronous behavior can insert a conditional delay in a program. The run method calls the heavyTask method that uses a lot of resources, we only want to start when these resources are available. We assume the systemLoadHigh method returns false when the needed resources are available. This is repeatedly tested by the busyWait handler. We use the same approach in concurrent programs except for the busy wait.

The language EScala was not designed with concurrent execution in mind, this can lead to unexpected behavior when using its event system via different thread. For example, the collection of handlers that are registered with an event can change while the same event is triggered. Unlike dedicated CEP systems, the language EScala cannot observe

```
class Synchronous {
  evt beginTask=before(heavyTask)
  def busyWait():Unit= {
    while (systemLoadHigh())
      sleep(1000) // 1 second
  }
  observable def heavyTask()= {
    // needs a lot of resources
  }
  beginTask += busyWait _
  def run()= {
    heavyTask()
  }
}
```

Figure 2.17 – Using synchronous events to wait.

event occurrences over time intervals.

2.4 Concurrency

When more than a single task is executed simultaneously, these tasks are said to be concurrent. This simultaneity may only be logical (pseudo-parallel execution) rather than physical (real parallel execution). For example, the execution of the tasks can be interleaved on a single processor.

The granularity of concurrent tasks can vary from programs for different users down to multiple tasks inside a single program. We focus in this thesis on the concurrent tasks inside a single program. An example of this kind of concurrency are programs that react to their Graphical User Interface (GUI), while performing other tasks for the user. Today there is an additional motivation for concurrency, since most processors offer multiple cores. In that case, the performance can benefit from concurrent execution of the tasks.

When the execution takes place in different parts of code at virtually the same time, we say that each part of the code is executed as a different thread.

In this section, we show different programming models that are traditionally used for concurrent programs. We provide examples in the Scala language, because this language is the basis for EScala, the language that we introduce in this work is based on EScala. While functional programming is not a concurrency model, we mention it

because its relation to the EScala event expressions and the implicit concurrency via futures.

Programming Models The design of concurrent programs can adopt different programming models. These models range from the shared-memory model to a high-level model that shares nothing. The high-level shared-nothing model shields the programmer from some of the complications associated with concurrency. In the shared-memory model threads can synchronize via the memory, while synchronization in the shared-nothing model is the result from message passing. Finally, we complete the overview by mentioning the transactional memory systems.

These models are supported by different programming languages, either as an integral part of the language or via libraries.

2.4.1 Synchronization through Shared Memory

In the shared-memory model, several tasks share their memory, or a part thereof, typically the heap. The current processors with multiple cores can execute in parallel tasks including access to the shared memory. Performance can benefit from this model, because exchanging data between tasks does not require copying data. However, to ensure consistency of shared data, the programmer needs to synchronize accesses to the shared data that is stored in the shared memory. A sequential program can swap the values in two variables by using an auxiliary variable. When two tasks use the same technique on the same variables, one expects the variables are restored to their initial values. When these tasks run concurrently the result depends on the interleaving of the instructions executed by both tasks. Such *race conditions* are prevented by synchronizing accesses to the data in the shared memory. However, the correct implementation of the synchronization is complex and therefore this model is error prone. Despite its complexity, the shared-memory model is widely used and it is supported by many programming languages and libraries.

Dijkstra describes semaphores [29] to control access to shared data. A semaphore is a counter implemented as an unsigned integer. Before starting an atomic group of actions on shared data the counter has to decrease by one. After completing the actions, the counter is incremented. Such a *binary* semaphore can be used as a lock. As a generalization, the counter can be initialized with n, it keeps track of how many threads can acquire that semaphore. The implementation of a semaphore relies on processor instructions that atomically test and change a value in memory.

A programmer who uses semaphores has to identify each part of the shared data that is

accessed by atomic group of instructions. Identifying the atomic groups is a responsibility of the programmer. Then the programmer defines a semaphore for each part of shared data. Before code accesses shared data, the execution of the code needs to wait for the semaphore that controls the data it accesses. Afterwards it needs to release that semaphore. The following questions rise: when can the semaphore be released? What is the optimal granularity for the memory, with a low risk for deadlocks and maximal concurrency?

Coarse grained monitors [58, 52] are easier to use than semaphores. Instead of individually controlling each group of instructions manipulating some share resource, a monitor controls all the operations associated to the shared resource and automatically guarantees mutual exclusion through locking. *Conditional expressions* for *conditional variables* can be used in order to synchronize tasks trying to *enter* the monitor in order to execute one of its operations. A survey of these different mechanisms can be found in [18].

Libraries like Pthreads [19] and ACE [110] support semaphores. After Concurrent Pascal [52] introduced support for monitors, other programming languages supported monitor as well. Widely used programming languages like Java [46] and C# [53] also offer monitors. In these languages, each object can actually be used both as a lock and a monitor.

2.4.2 Synchronization via Message Passing

The correct implementation of programs in the shared-memory model is complex, since we showed that the model puts the responsibility for the complex task of synchronization on the shoulders of the programmer. Since this makes the shared-memory model error prone, we study alternative models that release the programmer from this responsibility by avoiding shared memory.

Functional programming is not a concurrency model, however, we mention it because it does not support assignments. This means that variables are immutable: once initialized, their value cannot change. Functional languages like Haskell [64] can support a form of assignment in the realm of functional programming by using monads. Hybrid languages typically support mutability either by introducing references, as in OCaml [73], or by distinguishing between mutable and immutable variables, as in Scala [91]. In Scala, mutable variables are defined after the keyword `var`, while immutable variables are defined after the keyword `val`.

The communication in the shared-nothing model can only use shared-data when it is immutable, since data races are impossible for data that cannot change. When several

concurrent tasks access the same immutable data, this behaves the same as when each of these tasks would have its own copy of the immutable data.

Examples of models that are based on the shared-nothing model are the Actor model [56] and futures [7].

Actor model In the Actor model [56, 2], a task, called an *actor*, has its own local memory and communicates by sending *asynchronous messages* to other actors. There is no concurrency within an actor. However, two or more actors can execute concurrently. Similarly to pure Object-Oriented programming where objects are the only encapsulating entities, in the pure actor model actors are the only encapsulating entities.

Each actor has a single mailbox, from which it reads messages and reacts to them. The model does not impose any order on reading the messages from the mailbox. After receiving a message, an actor can react with three types of actions or a combination thereof:

— change its *behavior*, the term *behavior* refers to the internal state of the actor

— send asynchronously a number of messages to other known actors

— create and start new actors

Actors only interact by sending asynchronous messages to actors. These messages may include actor *names* in order to create new communication channels between actors.. However, these messages cannot contain references to any data, since this would lead to shared data.

Hewitt created the first actor language Plasma [55]. In the nineties, the telecommunication company Ericsson developed the programming language Erlang [6, 5]. Erlang was developed for programs that need to work without interruptions, in particular fault-tolerant telephone systems. Such applications require replacing code of a running program as a way to reduce service interruptions. Following the pure model, Erlang actors do not share data, which helps to recover a failed actor, or replace an actor with a new version.

Programming languages like Erlang are designed to support the pure model, but Object-Oriented languages (e.g. Scala [50]), too, can support the Actor model. However, languages like Java or Scala do not encapsulate the internal state nor methods of an actor. This makes the programmer responsible for respecting the model by only communicating via messages with an actor and avoiding shared references by copying data. *Actor objects* (*active* objects, associated to a thread, and implementing actors) coexist with standard, *passive*, objects. By restricting access to the internal state of an actor object, for example by using visibility modifiers like `protected` and `private`, the programmer prevents changes to the internal state of an actor except by the actor itself.

Mixing objects and actors puts the responsibility on the programmer to avoid calling methods on an actor. Libraries like Scala actors [50] can help the programmer with problems like managing threads and provide a syntax similar to an actor language like Erlang. Other frameworks are discussed in [17]. Another issue is that most implementations of the actor model in an Object-Oriented language do not copy messages to the mailbox of the receiver. This is not only motivated by the performance drawback that copying implies, but copying is not always possible. For example, how does one copy a file handle?

If the message is an object, whose reference is simply passed from sender to receiver the use of immutable values, as suggested by Haller and Sommers [50], is only safe from race conditions when the complete data structure is immutable. We can relax this total immutability by copying the mutable parts. Kilim [112] avoids race conditions by allowing only a single reference to the object used as a message. The sender abandons its references to the message object in favor of the receiving actor. More sophisticated schemes are discussed by Haller and Odersky [49].

Figure 2.18 shows an actor in Scala that maintains a single integer value as a buffer. The value can be set by sending the message `Put` and the set value can be retrieved through the message `Get`. The behavior of a Scala actor is defined by the method `act`, which typically looks for the possible messages, extracts their data (using pattern matching), and handles them. Actors are aware of the sender of each message, which makes it possible to `reply` to a message. The `!` operator sends an asynchronous message to an actor, using the `!?` operator instead waits to receive a reply after sending an asynchronous message, which results in a synchronous behavior.

Since Scala uses the JVM instead of a dedicated virtual machine like for example Erlang, threads are not always the best option. Because there are only a limited number of threads available, Scala actors can use *event-based* actors [48]. These actors do not require a dedicated thread, because the system provides them with a thread when they have messages waiting in their mailbox. The example shown in Figure 2.18 uses such an event-based actor. This uses the constructs `loop` and `react`, which releases the underlying thread after handling a message. A thread-based actor uses an infinite loop and the `receive` construct instead of `react`. The `react` construct in event-based actors can be considered as registering a handler to an event that indicates the reception of a message that matches the handler. There is an event for each type of message that the actor can react to. Each time such an event occurs, the actor runtime reacts by executing the handler via a worker thread. While an event-based actor does not require a dedicated thread, the programmer needs to prevent that a thread blocks while executing `react`.

```
1 import scala.actors._
2 import scala.actors.Actor._
3
4 case class Put(value:Int) {}
5 case class Get() {}
6
7 class Buffer extends Actor {
8   def act()={
9     var cell:Int=0
10     loop {
11       react {
12         case Put(x) => cell=x
13         case Get() => reply cell
14       }
15     }
16   }
17 }
18 def run()={
19     val anActor=new Buffer
20     anActor ! Put(5)
21     val res= anActor !? Get()
22 }
```

Figure 2.18 – Actors in Scala.

```
1 val f=future {
2           slowCalculation()
3         }
4 f.onSuccess {
5    case x => println("the␣result␣is␣"+x)
6 }
7 // go on with other tasks
```

Figure 2.19 – Future in Scala.

Futures Futures can offer the programmer concurrency in an implicit way. Originally, futures [7] were introduced as call-by-future and considered an alternative for call-by-value and call-by-reference. The terms *future*, *promise* [41], *delay* and *eventuals* [57] are often used as synonyms. Futures in current languages like Scala [91] are encapsulated computations that are executed by another thread than the one that defined them. The definition of a future returns a placeholder for the result of the calculation. Once the result is available, the placeholder gives access to this result. When the program requires a result that is not yet available, the program waits until result becomes available.

Figure 2.19 shows a Scala `future` with an `onSuccess` handler, optionally an `onFailure` handler can be defined for a future, failures can express calculations without a result or an exception.

2.4.3 Transactional Memory

While locking in shared memory is pessimistic since it expects race conditions to occur frequently, Transactional Memory [54] has an optimistic strategy by expecting the conflicts are not likely to occur, and that they can be solved afterwards. Similar to database transactions, threads access memory that is isolated from other accesses, after completing a transaction, it can either be successful and atomically modifies the shared memory, or if there is a conflict it is possible to restart a transaction from the current situation. This model presents to the programmer memory that is not shared with other threads, which is similar to sequential programs. However, the programs that use transactional memory often interact with users or parts, that are not managed by the same transaction system, then the implementation needs to buffer the communication to simulate transactions. These buffers can require a lot of memory. When transactions communicate, the transactions cause other transactions that depend on their result, which makes efficient implementations hard.

The early versions of transactional memory relied on special hardware. Currently, there

are software implementations that do not rely on such dedicated hardware. These implementations are known as software transactional memory (STM). Exploring this model could be useful. However, in this thesis we only consider a pessimistic model, although we aim to relieve the user as much as possible from the low-level locking and its associated complexities.

2.5 Joins

In this section, we discuss an alternative for the classic concurrency models. We begin introducing process calculi, to show that the π-calculus [85] is not the only option. In this work, we use elements from the Join Calculus that we briefly introduce in this section. We show how these elements work for a user and how these ideas are part of several languages. Finally, we show examples of implementations with Joins in a hypothetical Scala-like language. By using a hypothetical language that is based on Scala the reader can compare with other languages that we show in this section, and also get further acquainted with the language Scala.

2.5.1 Calculi for concurrent programs

From the three models described in Section 2.4, we focused on the shared-nothing model, since this is mature and avoids complexities for the programmer. Now we look to the formal underpinnings for concurrency, first the widespread π-calculus [85], followed by an alternative, the Join calculus [39], which targets distributed computing,

Sequential programming is often studied in a formal way by using the λ-calculus [8] (lambda calculus), which is a formalism for functional computations. It becomes a simple programming language by extending it with primitives that encode numbers, functions, booleans, etc.

2.5.1.1 Pi Calculus

The π-calculus (pi-calculus) is a widespread formalism for concurrent computations. However, it is not the only member of the group of process calculi and algebras (CSP [59], CCS [84], ACP [10], LOTOS [123]) that formally describe concurrent computations.

Like the other members of that group, the π-calculus describes concurrent processes. This calculus can describe processes that dynamically reorganize the communication channels between each other. Processes can send and receive data via channels, that

can be created by each process. The π-calculus [85] treats variables and constants as the same and refers to them by a name. Names are also used to refer to communication channels. Therefore, it is possible to send the name of a communication channel via another channel. This does not only makes it possible to describe changing connections, it is also a basic computation step. Similar to λ-calculus, the π-calculus is Turing complete, which can be shown via bisimulation, i.e., encodings of the λ-calculus in the π-calculus and vice versa.

2.5.1.2 Join Calculus

The Join calculus [39] is designed to support locality and asynchronous communication. Which makes distribution possible, since it does not have to use expensive remote channels as an intrinsic part of a computation. Before explaining this calculus, we show the weakness of the π-calculus with respect to distribution.

Sending a channel via other channels, is specific to the π-calculus. This is not only the basic computation step in this calculus, it supports distribution and mobility. The communication uses channels that belong to an *aether*, in other words they are shared among the distributed parts. However, communication via a channel to remote location is more expensive than local communication. Since exchanging channels is a basic operation in the π-calculus, the expensive communication to remote channels is hard to avoid.

While the sequential calculations are sometimes represented by mechanical machines like SECD [71], concurrent calculations can be represented by other metaphors like the chemical abstract machine [11]. These machines represent calculations as chemical reactions, which can take place concurrently. The Join calculus can be modeled by using a variant of a chemical abstract machine, the reflexive chemical abstract machine (rCHAM). Molecules represent asynchronous events or processes. We consider a reaction tube that contains a liquid, which represents the imaginary chemical soup. There are channels that each inject a single molecule into the soup. Inside the reaction tube, a reaction according to rules provided by the programmer takes place. This reaction reorganizes the injected molecules into new molecules. The word *reflexive* refers to the possibility to add new molecules and new rules.

Formalization Formally, the Join calculus consists of processes, definitions, patterns. Lévy [74] describes the formal concepts as shown in Figure 2.20. The Join patterns (J) are related to the chemical reactions in the chemical model, these patterns are combined into a definition.

$P ::= \mathtt{0}$	empty process
$c(\tilde{d})$	emission of $\tilde{d}$ on c
$P_1 \& P_2$	parallel composition
$\mathtt{def}\ D\ \mathtt{in}\ P$	definition
$D ::= J \triangleright P$	reaction
$D_1 \| D_2$	disjunction
$J ::= c(\tilde{d})$	reception of $\tilde{d}$ on c
$J_1 \& J_2$	synchronization

Figure 2.20 – Formalization of the Join calculus.

Join Calculus operates on channel names (c and d) in Figure 2.20. The notation $\tilde{d}$ represents a tuple of names $d_1, \ldots, d_n$.

Processes (P) can send data $\tilde{d}$ over a channel c. A *Process* can be a parallel composition of zero or more processes. *Definitions* can be used in a *Process*.

A primitive *Definition* contains a single pattern.The *choice* operator (|) composes definitions into *disjunctions*. This operator indicates that evaluating a disjunction requires a choice.

Join patterns observe the data available via one or more channels. When data is available via all channels of a Join pattern, the pattern reads the data from all its channels in an atomic action, which makes the received data available to the reaction that starts because the pattern matched. The reaction is a process. The definition of processes is flexible enough to start zero or more processes. In the rest of this section, we use the word *event* when data becomes available via a channel. In general, Patterns are *linear*. This means they observe different channels. However, the actor language JErlang [97] supports non linear patterns.

The Join patterns in a disjunction observe the channels concurrently. Therefore, patterns test and read data from all their channels in an atomic operation. When within a disjunction, two or more Join patterns observe the same channel, these patterns can compete for events from the shared channel. This competition occurs when two or more Join patterns lack an event from the same single channel, the choice of the Join pattern that matches is non deterministic because, conceptually, these pattern test at the same time whether they can match and atomically retrieve the data via the channels that each pattern observes.

```
1 def a() & b() = d()
2  or a() & c() = e() ;
```

Figure 2.21 – Example of a Join pattern.

Disjunction of two patterns:
$\underbrace{(a\&b)}_{Pat_1} \mid \underbrace{(b\&c)}_{Pat_2}$

JEScala notation:
$(Pat_1, Pat_2) = (a\&b) \mid (b\&c)$

(a)

Input	Pattern
a, b	1
b, a	1
b, c	2
c, b	2
a, c, b	$1 \vee 2$
b, a, c	$1 \vee 2$
c, b, a	1

(b)

Figure 2.22 – Example of competing patterns.

Figure 2.21 illustrates a disjunction that contains two Join patterns using a syntax that is fit for a programming language. The letters `a`, `b`, `c` are the names of channels. Since the names are followed by an empty pair of parenthesis, they receive data without any value. The binary Join pattern (`(a() & b()`) joins the channels `a` and `b`. The empty pairs of parentheses indicate that these channels receive data, that does not carry any values. When data is received from each channel an undefined process `d()` starts. Similar to the channels the empty pair of parentheses indicates the absence of values. In the JoCaml notation, the $\triangleright$ is replaced by the equal sign (=). The definition of a second Join pattern that combines `a` and `c` follows after the `or` that replaces a vertical bar (|). Both patterns share the channel `a`. When data is available at both channels `b` and `c`, the data arriving via channel `a` can complete both of these patterns, which leads to the non determinism described above and results in either starting the process `d()` or `e()`. Some applications of Join patterns rely on this non determinism in selecting between possible matching patterns in a disjunction.

Disjunctions in JoCaml create the channels to receive data. This implies that each channels has a unique receiver. This receiver is a disjunction which contains one or more patterns.

Interactions inside a disjunction We explain the interaction among Join patterns with the disjunction in Figure 2.22a, which contains two patterns Pat_1 and Pat_2. The languages that we discuss further in this section assign an action to both Pat_1 and Pat_2.

```
1 def (a & b) = c ;
2 def (d & e) = f ;
3
4 def (a & b) = c
5  or (d & e) = f ;
```

Figure 2.23 – Combining Join patterns.

This action is a piece of code like a function, pattern matches, its action executes. However, this action can be an empty block. For the Object-Oriented languages this action is a method body. We introduce later the language JEScala. For now, we only show its notation of a disjunction with two binary Join patterns.

Figure 2.22b shows for the input via the channels from the first column the resulting pattern in the second column. For example, receiving first data via channel a and then via channel b triggers the first pattern. When the events arrive in the reverse order via the same channels, this triggers the same pattern. When repeating this with the channels (b and c) from pattern Pat_2, that pattern matches.

Join patterns compete when they both need to receive data via a shared channel to be complete and match. This situation occurs when data is first sent via the channels a and c, each pattern can match after receiving data via channel b. Then both patterns compete, the pattern that grabs the event first matches. In the pure model, the selection of the winning pattern is a non deterministic choice. However, there are deterministic variants in languages like JErlang [97] and optionally Join Java [60], these languages – which we discuss further in this Section – try the patterns in the same order as they are defined. In Join Java the programmer can decide the matching order.

Merging disjunctions A disjunction combines interacting Join patterns. Adding a Join pattern to a disjunction can change the interaction among the existing patterns in unexpected ways. However, when the additional Join pattern does not share any channel with the other Join patterns, it does not interact with the other Join patterns.

This makes it possible to combine the Join patterns of two disjunctions, when there are no channels that are part of both disjunctions. The first two disjunctions in Figure 2.23 do not share any channel, which makes it possible to write a single disjunction that combines these two patterns without changing their behavior.

Combining and splitting disjunctions can help the programmer to organize code. In Join languages that use an object as an implicit disjunction, combining unrelated patterns in a single object can help to keep parts that belong together in a single object.

Synchronous channels Channels in the Join calculus can be either synchronous or asynchronous, the synchronicity decides whether sending to such channels blocks the sender or not.

Fournet and Gonthier [39] describe what we consider to be the basic calculus. Its channels are *asynchronous*. With only asynchronous channels, Join patterns are the only way to synchronize. This basic Join calculus can encode the usual constructs that control the flow of the execution, in particular synchronously sending data with a reply and sequential composition. Typically this encoding uses explicit continuations but languages built on top of the Join calculus can hide these encodings by introducing *synchronous* channels.

Asynchronous channels can encode synchronous channels by using continuation passing style [107]. The encoding of a synchronous channel using an asynchronous channel relies on a continuation that receives the data from the encoded synchronous channel. When the data sent to the asynchronous channel includes the continuation. When a pattern that observes the asynchronous channel matches, the received continuation is executed as part of the reaction and the code after the emission to the synchronous channel continues.

2.5.2 Join Languages

Join patterns and asynchronous events are the elements that are part of programming languages that we refer to as *Join languages*. Join languages represent channels from the Join calculus in different ways. Some Join languages use the word *event* to refer to data sent via a channel or for the channel itself. However, none of the Join languages that we discuss here, supports events with implicit invocation, the events from a Join language are only received by a single entity (disjunction) that refers to the channel on which the event is sent. We show the implementation of a buffer in each of the languages.

Integrating elements from the Join calculus in a language is possible by changing the implementation language (preprocessor, compiler, interpreter, ...) or via an additional library. Languages with a modified compiler can analyze the use of Joins and generate code that is optimized for a specific situation. A modified language can provide the user a consistent syntax, which libraries cannot always provide. For example, implementing events as methods via a library is hard.

JoCaml The first language designed to support the Join calculus was JoCaml [23]. While the implementation of JoCaml is a modification of OCaml, the examples for

```
1 def get() & put(s) = reply s to get;;
```

Figure 2.24 – Buffer with JoCaml.

JoCaml do not rely on the support for Object-Oriented programming of OCaml. The current implementation [79] of JoCaml sacrifices distribution and mobility for a better integration into standard OCaml.

Join definitions in JoCaml begin with the keyword `def`. As defined by the Join calculus, a join definition combines reactions. When a process replies to a channel, the channel is synchronous. For the programmer, synchronous channels in JoCaml behave like a functions.

Figure 2.24 shows a buffer in JoCaml. The join definition on Line 1 includes a single Join pattern that results into a primitive process. A Join definition can define more than one Join pattern together with its resulting processes by separating them with `or`, hence the term *disjunction*. The example shows on the left-hand side of the equal sign (=) a pattern with the channels `get` and `put`. An ampersand (`&`) at the left of the equal sign is the separator between channels, whereas an ampersand at the right-hand side would separate the processes to start. The `put` channel is defined with a single argument. The process at the right uses the same argument to send its value via the `get` channel. JoCaml supports both synchronous and asynchronous channels. A channel is synchronous when a process at the right side replies to it.

Funnel Whereas JoCaml, which closely follow the initial definition of the Join calculus is close to Caml, Funnel [90, 89] is close to Scala, which was developed afterwards. This is made possible by building Funnel on a small variant of the Join calculus, the *object-based Join calculus*. The main difference is the introduction of *qualified definitions*, where channel names can be built from names concatenated with periods. This small addition is sufficient to make the *object-based Join calculus* a low-level language into which both the function and object-oriented constructs of Funnel can be translated.

```
1 def get:T & put (x:T) = x
```

Figure 2.25 – Buffer with Funnel.

Figure 2.25 shows the buffer in Funnel. Similarly to JoCaml, a definition starts with `def` followed by the channel descriptions composed with an ampersand. These descriptions look like function or method headers, they define the name of the channel and between parentheses the data received via the channel. The first channel, `get`, is synchronous

and returns a value of a type T, the channel put receives an argument x of the type T. Channels in JoCaml are synchronous when a resulting process replies to that channel. Instead, Odersky [89] mentions that in Funnel only the first channel of a pattern can return a value, which means that except possibly the first channel, all channels are asynchronous.

The Join definition in JoCaml and Funnel is similar to an object with methods. The languages Join Java [60] and Polyphonic C# [9] use objects as Join definitions (disjunctions). In both of these languages, Join patterns can contain only a single synchronous channel.

```
1 class Buffer {
2    String get() & put(String s) {
3       return s;
4    }
5 }
```

Figure 2.26 – Buffer with Join Java.

Join Java The integration of Join patterns into Java resulted in the language Join Java [60]. The patterns are defined by methods with multiple headers that are composed with an ampersand (&). Similar to other methods in Java, these patterns have a body. The body has access to the actual parameters from each of the headers. Like in Funnel, only the first method header can be synchronous. In that case it specifies the type of the value returned by the body. The other headers are always asynchronous, therefore they do not define a return type. Normal methods that have a single header and define signal as return type are executed concurrently with their caller. In other words, the caller does not wait for the method to return. Figure 2.26 shows a buffer, which is implemented as a Join Java class. Calling the synchronous get method waits until a call to the method put is received as well. The calls to the put method do not wait, because that method header is not the first in the pattern. The return statement in a body can only return a result via the method call of the first header, since only that header defines a return type.

Join Java combines all Join patterns of an object into an implicit *disjunction*, while JoCaml and Funnel use a dedicated construct to explicitly group Join patterns. Both languages have a def construct for a disjunction, the figures do not show that JoCaml composes patterns in a disjunction with the or keyword, whereas Funnel separates them with a comma. When a single method call can cause a match in two patterns, the Join calculus suggests a non-deterministic choice. However, the modifier ordered can mark a class, which results in testing the patterns in their order of definition.

```
1 public class Buffer {
2    public string Get() & public async Put(string s) {
3       return s;
4    }
5 }
```

Figure 2.27 – Buffer with Polyphonic C#.

Polyphonic C# Similarly to extending Java with Joins, C# [53] can be enriched with concepts from the Join calculus, which results in Polyphonic C# [9]. Similar to Join Java, this language does not integrate all concepts from Join calculus. The resulting language is similar to Join Java mentioned above. This language uses objects as disjunctions. A method headers defines a channel with it receiving arguments. Patterns use ampersands (&) to compose method headers. A pattern with two or more headers has a single body. Polyphonic C# calls this combination of method headers with a body a *chord*. Asynchronous methods specify `async` as their return type. Similarly to Funnel and Join Java, there can be at most one single synchronous header in a pattern, but it is not always the first in a pattern.

The authors motivate this limitation in two ways. First, C# uses the `return` keyword for the result of a method while only an synchronous method header can specify a return type, since there is only a single return there can be only on synchronous header. Second, the thread that calls the synchronous method executes the body. For some libraries, like user interface frameworks, it is important to access the library always with the same thread, which is provided by the synchronous method call. Since a *chord* can only have a single method header with a return value, the thread that called this method executes the method, similar to normal methods with a single header.

In the Join calculus, Join patterns concurrently try match. This leads to non determinism when a method call can cause a match in more than one Join pattern. Unlike Join Java, Polyphonic C# does not offer any alternative matching strategy.

Cω[106] (C omega) is a research extension of C#, which integrates all extensions from Polyphonic C# and other extensions like Linq [121] for data access. Unlike in Polyphonic C# chords, the synchronous method can only be the first header of its Join pattern.

Joins Library The first library implementation for Join patterns is the Joins library [108] for the .NET platform. Figure 2.28 shows the buffer example implemented with this library. Implementations based on a modified language can express channels as methods. When relying on a library, this is not possible as there is no way to provide

```
1  public class Buffer {
2      public Synchronous<int>.Channel Get
3      public Asynchronous.Channel Put<int>
4      public Buffer() {
5          Join j = Join.Create();
6          j.Initialize(Get);
7          j.Initialize(Put);
8          j.When(Get).And(Put).Do(delegate (int i) {return i })
9      }
10 }
```

Figure 2.28 – Buffer with Joins Library.

methods with multiple headers. Channels have to be implemented as objects. In the Joins library, these objects are instances of the classes `Synchronous.Channel` and `Asynchronous.Channel`. Calling `Join.Create` generates a disjunction. For each channel the method `initialize` of the created `join` object needs to be called. Defining patterns within this disjunction means calling the `when` method followed by `and` methods. The specification of a pattern starts by calling the `when` method for the first channel followed by repeated calls to the `and` method. Calling the `Do` method with a *delegate* argument defines the body and terminates the pattern definition. A *delegate* is a C# construct to support anonymous functions. A later implementation [122] with the same Application Programmers Interface (API) was used to study optimizations like fine-grained locking.

Concurrent Basic Concurrent Basic [109] integrates the aforementioned joins library into the language Visual Basic. In addition to this integration, this language supports inheritance. Figure 2.29 show the implementation of the running Buffer example. The `Buffer` has a type parameter `T`. Since `Get` and `Put` are used in this language, we use `MGet` and `MPut` instead. Line 2 defines the asynchronous `MPut` channel with an argument of type `T`. Line 3 defines the synchronous `MGet` channel, which returns values with the same type `T`. The pattern is defined as the function `Pattern` followed by the keyword `When` and a comma separated list of channels. The name of the `Pattern()` function is not used in the rest of the code but when comparing this code with the Joins library implementation in Figure 2.28 the name of the function is required for the delegate used by the library.

The `SubBuffer` subclass defines the asynchronous `MAltPut` channel (Line 11). In this example, this channel is the alternative for the `MPut` channel in the class `Buffer`. The `SubBuffer` class adds a Join pattern (Line 12) to the one defined in its super class. This

```
1 Class Buffer(Of T)
2   Public Asynchronous MPut(v as T)
3   Public Synchronous  MGet() as T
4   Private Function Pattern() When MPut,MGet
5     Return me.v
6   End Function
7   End Class
8
9 Class SubBuffer
10   Inherits Buffer
11  Public Asynchronous MAltPut(v as T)
12  Private Function AddPattern() When MAltPut,MGet
13     Return me.v
14  End Function
15 End Class
```

Figure 2.29 – Buffer and a subclass with Concurrent Basic.

```
1 class Buffer extends Joins {
2   val Put = new AsyncEvent[Int]
3   val Get = new SyncEvent[Int]
4   join {
5       case Get() & Put(x) => Get reply x
6   }
7 }
```

Figure 2.30 – Buffer with ScalaJoins.

pattern composes the `MAltPut` channel with the `MGet` channel, which is inherited from the super class.

ScalaJoins The focus of the ScalaJoins [51] library is the integration of Join patterns into Scala. Figure 2.30 shows our running buffer example implemented in ScalaJoins. Similar to the Joins library for .NET, ScalaJoins represents channels by objects, that are instances of the classes `SyncEvent` and `AsyncEvent`. Join patterns are defined by `case` instructions in a `join` block. A `case` instruction uses the `&` operator to compose the events that are defined in the same class.

A `join` block is not an additional Scala construct, it is implemented as a method `join`

```
1 receive {get, X} and {set, X} -> {found, X} end
```

Figure 2.31 – JErlang buffer.

in Joins *trait*. A Scala *trait* is comparable to an interface in Java, since both define method signatures. However, traits can also contain implementations. The order they are included in a class or object is important, since a method body can override a previously defined method. The argument of the join method is a *partial function*. Partial functions in Scala are not defined on their entire domain. In Scala, such functions are usually a block with case statements. ScalaJoins uses a case statement to express a join pattern. The composition operator (&) is implemented as a method for all event classes. The reaction for a matching pattern is defined after the arrow (=>) of the case instruction.

For a simple implementation, the semantics of the asynchronous events is different to the one of asynchronous method headers in Polyphonic C#. Its asynchronous events do not block their delivering thread in a disjunction, except when the last event that matches a Join pattern is asynchronous, its thread executes the resulting block of code. However, this simple implementation can cause unexpected behavior. A programmer does not expect that triggering asynchronous events blocks, but this happens when a Join pattern is composed with only asynchronous events and the resulting code does not return immediately. Since the code that triggered the last asynchronous event in the pattern waits for this reaction to return. The join method is implemented as a setter for a variable defined in the Join trait. The last call of the join method defines the disjunction that is used for the instance.

The ScalaJoins library also contains an integration of Joins into Actors. This integration makes it possible to receive and react to a pattern of messages instead of reacting to a single message once it is received. Since actor messages are always asynchronous, it is important that the sender and receiving actor agree on the synchronization protocol, which relies on answering to messages.

JErlang JErlang [97] extends the actor language Erlang [5] with Joins. Figure 2.31 shows a buffer implementation in JErlang. Messages between actors are always asynchronous, but a synchronous semantics is possible by waiting for a reply. This language is implemented in two ways. First, a library can be used with existing Erlang systems. Second, a modified virtual machine as a faster implementation. Unlike Join patterns in other Join languages, the events or channels observed by a Join pattern do not have to be unique. This makes it possible to extend the running buffer example to reply a get message with a combination of the received values from two put messages.

```
1  spawn {
2    sync get() and async put(x) {
3      return x;
4    }
5    body {
6      while (true)
7        serve;
8      }
9    }
10 }
```

Figure 2.32 – Buffer in JCThorn.

Unlike the pure model for the Join calculus that selects matching patterns in a non deterministic order, JErlang selects matching patterns in the order that they are defined, similar to ScalaJoins. Different from the other languages discussed here, JErlang supports timeouts when waiting for missing events for completing a pattern.

JCThorn Thorn [12] is a scripting language that supports concurrency and runs on the JVM. JCThorn [94] combines the Thorn language with elements from the Join calculus. Since this language is a hybrid, it is different from Object-Oriented Join languages. The programmer defines *components*, which are similar to actors. This language uses the Object-Oriented elements from Thorn to build the code inside these *components*. The messages are the foundation on top of which high-level communication is built. Using this high-level communication to interact with a *component* looks similar to calling method. While it is possible to build Join patterns based on the low-level messages, there is a way to combine the method-like constructs into patterns with a body similar to methods in other Object-Oriented Join languages. Figure 2.32 shows the JCThorn implementation of the running buffer example. The Join pattern combines the methods `get` and `put` into a pattern (Line 2) and defines a body that returns, like a method call. The `body` block (Line 5) executes an infinite loop similar to an actor to receive and handle high-level requests using the `serve` (Line 7) statement.

Figure 2.33 summarizes the Join languages that we discussed. Column 3 shows for each of the discussed languages which language it extends. The fourth column shows how channels are implemented. ScalaJoins uses events that are only received by a single disjunction, comparable to the channels from the Joins Concurrency Library. Column 5 shows the synchronicity supported by each language, it is still possible to build synchronous communication on top of asynchronous channels by waiting for a reply. The representation of a disjunction in shown in Column 6, some language unify a disjunction

	Language	Channels	Sync	Disjunction	Matching
JoCaml	Caml	Function	Both	Explicit	Non deterministic
Funnel	Funnel	Function	Both	Explicit	Non deterministic
Polyphonic C#	C#	Method	Both	Object	Non deterministic
Join Java	Java	Method	Both	Object	Both
ScalaJoins	Scala	Imper. Event	Both	Explicit	Non deterministic.
JErlang	Erlang	Message	Async	Actor	Deterministic
JCThorn	Thorn	Message	Async	Component	Deterministic
Join Conc. Lib.	.NET	Channel	Both	Explicit	Non deterministic
Concurrent Basic	Visual Basic	Channel	Both	Object	Non deterministic

Figure 2.33 – Languages implementing Join abstractions.

with the encapsulation defined as an object or an actor, others use an explicit construct instead. Column 7 shows how a pattern is selected when more than one pattern matches, most Join Java provides the programmer with a choice between the non deterministic choice of the chemical model and the order the patterns are defined. Chapter 9 completes this table.

2.5.3 The role of Joins in application design

Join languages integrate two constructs. First, the asynchronous channels, which do not block their sender. Second, the Joins that synchronize the reception of the data that they receive. In Section 2.5.2, we gave an overview of languages, like JoCaml [23], Polyphonic C# [9] and Join Java [60], that support these constructs.

The observation that other languages integrate elements from Join calculus is already an indication that these elements are useful additions to a language. Some of these languages, like Join Java and Polyphonic C#, implement examples to show that asynchronous events and Joins are useful extensions. In the same way, we could show a selection of examples in the language that we present in this work. However, we need to introduce several concepts of this language before showing such examples. Therefore, we use for now an hypothetical language based on Scala with the extensions that Polyphonic C# adds to C#. This approach has two advantages. First, we can skip for now an explanation of our language. Second, the reader can get more familiar with Scala.

We selected examples that are used further in this thesis as examples or in case studies. These examples show that combining patterns in a disjunction is useful when implementing concurrent applications. Writing similar applications with other techniques is

```
class Lock {
   def lock():Unit & free(arg:Int):async {}
   free() // constructor
}
class LockClient(val lock:Lock) {
   def criticalRegion={
      lock.lock()
      ...
      lock.free()
   }
}
```

Figure 2.34 – Lock with Joins.

of course possible. However, semaphores or monitors require mixing synchronization code with code that is responsible for the desired functionality.

None of these examples can benefit from implicit invocation since we use for now a language that only supports method calls, which is similar to Join Java or Polyphonic C#. Some use cases for the complete language (Chapter 4) rely on the finite state machine (FSM) example that we explain here by using a Join language without an advanced event system.

Similarly to Polyphonic C#, methods that define `async` instead of a return type do not make their caller wait until they return. Objects implement a single implicit disjunction of methods with multiple headers separated by an ampersand (`&`). Our synthetic language respects the limitation of at most one synchronous method header in a pattern, similarly to Polyphonic C#. While Scala does not requires parentheses for defining or calling methods without arguments, we show them for the convenience of the reader. In addition, we omit the equal sign between the headers and the body, in order not to suggest that the last header is the synchronous one, the one that can return a value.

Figure 2.34 shows a lock that is implemented by a single Join pattern without a body. In Scala, instructions in the body of a class are executed as a constructor, Line 3 calls the asynchronous `free` method as part of the constructor. The Join pattern on Line 2 waits for a synchronous `lock` method call. Once this call arrives, the waiting asynchronous `free` method is absorbed by the pattern. The next call to `lock` waits, since there is no `free` method call waiting to match with it. Before entering a critical region, each client of the lock has to call the synchronous method `lock` (Line 7) and wait for it to return. During the execution of the rest of the body, there is no method call to `free` waiting in the lock. Calling the `free()` (Line 9) method when leaving the critical region releases

```
1 class Actor {
2   def start():async=while (true) { threadReady() }
3   def threadReady():Unit & msgSay(arg:String):async {
4     println("Say␣␣"+arg)
5   }
6   def threadReady():Unit & msgMark(arg:Int):async {
7     println("Mark␣"+arg)
8   }
9 }
```

Figure 2.35 – Actor with Joins.

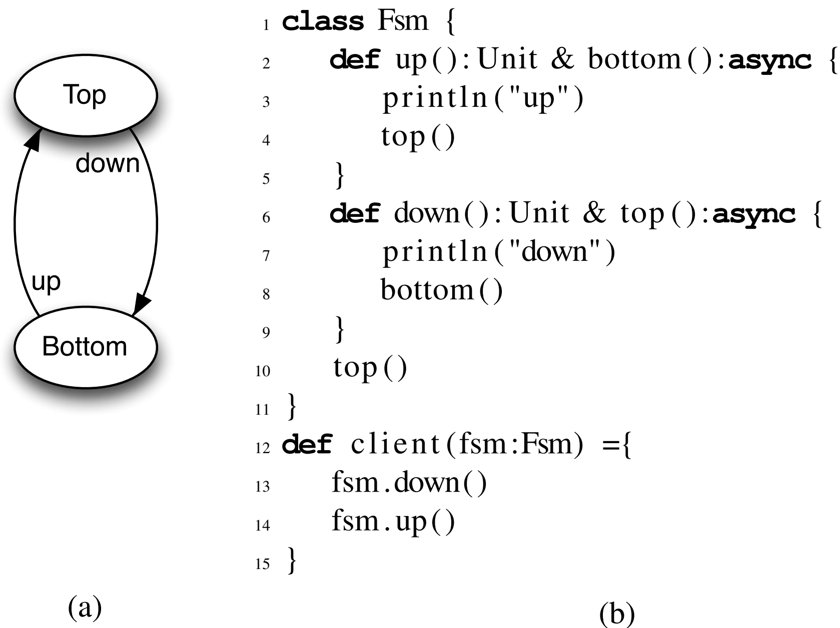
the lock.

The clients are responsible for calling the `free` method only after calling the `lock` method. The invariant of this implementation of a lock is that the number of waiting free calls is always zero or one. A single waiting `free` call represents the state `free` of the lock.

Figure 2.35 shows the implementation of an actor. The asynchronous `start` (Line 2) method makes the actor active, which makes it handling the messages it receives. Of course starting the actor can be part of its constructor, instead of an explicit `start` method like the Scala actors. The body of the `start` method is an infinite loop that calls the synchronous method `threadReady` and waits for it to return. Both Join patterns (Lines 3, 6) observe the method calls to `threadReady` and an asynchronous message method (`msgSay`, `msgMark`). Clients of this actor send a message to the actor by calling one of its asynchronous message methods. When a `threadReady` call is waiting and a client sends the message `msgSay` with a string argument, the client does not wait because the method header of the asynchronous `msg` method is marked as such by returning the `async` type. Concurrently with the caller that continues, the body prints a message. After this, the `threadReady` method returns without a value and the loop in the asynchronous `start` method begins another cycle.

A Join-based actor behaves differently from a Scala actor [50] and an Akka actor [47]. Each message has its own queue, the selection of waiting messages happens in a non-deterministic way, while Scala actors store all messages in a single queue, to handle the messages in the order they were received.

The lock example in Figure 2.34 represents the free state with a waiting call to the `free` method. We extend the same idea to multiple states in Figure 2.36b to implement the Finite State Machine (FSM) shown in Figure 2.36a. It is another example that uses waiting calls to represent states. This example represents the state by a waiting call to




```
1  class Fsm {
2    def up():Unit & bottom():async {
3      println("up")
4      top()
5    }
6    def down():Unit & top():async {
7      println("down")
8      bottom()
9    }
10   top()
11 }
12 def client(fsm:Fsm) ={
13   fsm.down()
14   fsm.up()
15 }
```

Figure 2.36 – Finite State Machine (a) implemented with Joins (b).

the asynchronous method `top` or `bottom`.

The first Join pattern (Line 2), can only match when there is a call to the `bottom` method waiting. The other Join pattern can match in case of a waiting `top` method call. Each asynchronous method triggers the pattern that corresponds to its underlying state. In the `bottom` state only the call to the `up` method can cause a match. The body of the first pattern prints the action and calls the asynchronous `top` method to set the new state, because the waiting call to `bottom` was consumed. Now only the second pattern (Line 6) can match when the `down` method is called. The call to the `top()` (Line 10) method at the end of the constructor initializes the state machine. When a pattern consumes the waiting call, its body calls an asynchronous state method to represent the new state. This results in the invariant that when the state machine is in a stable state, there is always a single pending *state* method call. This corresponds to the invariant of only a single state for a state machine at any time.

In the chemical model of the Join calculus, when several join patterns in the same disjunction match, a non-deterministic choice selects one pattern. This non-determinism can be used to schedule mutually exclusive tasks, for example, when they access both the same critical section. Figure 2.37 shows two tasks (`task1` and `task2`) that clients can execute. Each of these task methods is part of a Join pattern (Line 2 and 5) that combines an asynchronous method for each of the tasks with a synchronous `execute` method. Their clients want to execute (Line 10) `task1` or `task2` without waiting for

```
1  class Scheduler {
2     def execute():Unit & task1():async {
3       ..
4     }
5     def execute():Unit & task2():async {
6       ..
7     }
8  }
9  def client1(scheduler:Scheduler) {
10    scheduler.task1()
11 }
12 // Scheduler
13 while (true) {
14    scheduler.execute()
15    sleep(1000) // 1 sec
16 }
```

Figure 2.37 – Scheduler with Joins.

them to complete. A scheduler runs concurrently with such clients. This scheduler repeats calling the synchronous `execute` method and waiting after that during one second, before starting the next iteration. In our example, two patterns contain the synchronous `execute` method. When requests for both `task1` and `task2` are waiting, a non deterministic choice is made. Some implementations use a random generator for this choice and depend on its properties, such as fairness and uniformity. When only requests for either `task1` or `task2` are waiting, a waiting task of that kind is executed.

The two Join patterns (Lines 2, 7) both contain the asynchronous `counter` method with an argument that contains the current value of the counter. The body of each pattern calculates the new value (`w`) by either adding an argument `d` to the current value or subtracting an argument `d` from the current value. Then the body of the selected pattern calls the asynchronous `counter` method with the new value (`w`) and returns the new value. The counter is thread safe, since there can be only a single waiting call to `counter` method, which is consumed before calling the method with the new value. The initial waiting call is triggered by the constructor of the counter, the method has to be considered private. Clients, like for instance the `run` method (Line 15), can concurrently call the methods `inc` and `dec`, while the counter remains consistent.

```
1  class Counter(initial:Int) {
2    def counter(v:Int):async & inc(d:Int):Int {
3        val w= v+d
4        counter(w)
5        w
6      }
7      def counter(v:Int):async & dec(d:Int):Int {
8          val w= v-d
9          counter(w)
10         w
11     }
12     counter(initial)
13 }
14 ..
15 def run(cnt:Counter) {
16     while (true) {
17         val d= random.nextInt(10)
18         val n= if (random.nextBoolean) cnt.inc(d)
19                 else cnt.dec(d)
20         println("value␣"+n)
21     }
22 }
```

Figure 2.38 – Thread-safe counter with Joins.

2.6 Conclusion

In this chapter, we discussed aspects, events and concurrency. Relations between these topics were studied in pairs. First, Steimann et al. [114] related concepts from AOP with EBP. Further we mentioned languages that use simple events for aspects like Ptolemy, JasCo and EventCJ. In addition we discussed EScala as an example that integrates an advanced event system with aspects into an Object-Oriented language. Second, the relation between aspects and concurrency was studied by Douence et al. [30], who build on top of Event-Based AOP [31, 33]. However, this work only considered *implicit* events. These events are not integrated within a programming language but model at an abstract level the transitions of a labeled transition system [1]. Finally, concurrency in event systems is common in Message Oriented Middleware that connects multiple computers.

In this work, we use the same definition for asynchronous events that we used for asynchronous channels in the Join calculus. This means triggering an asynchronous event allows the program to continue. However, other languages use other definitions for asynchronous events. For instance, Panini [75] has asynchronous events, but it means that its handlers can execute concurrently. Asynchronous events enable the programmer to use concurrency without relying on explicitly managed threads. Since sending over an asynchronous channel does not block the sender, it results in concurrency between the sender and the observer of an event.

In the rest of the work, we combine the following benefits from the three fields we described before. Aspects can use implicit events that are part of the event system implemented in EScala. We combine this event system with ideas from the Join calculus to build a language that makes coordination possible among concurrent components.

Problem Statement

Contents

3.1 Case Study 80
3.2 Instrumentation 80
3.3 Coordination Logic 81
3.4 Discussion 84

In this chapter, we discuss the deficiencies in the design of applications that use existing Join languages. For this, we use a case study. The work is mainly based on EScala and JEScala. Both these languages are built on top of Scala. In Section 2.5.1.2, we use a hypothetical language that extends Scala in the same way as Polyphonic C# extends C#. We will call it Polyphonic Scala and develop it further in this chapter. The advantage of using this language is twofold. First, since it is based on Scala, the reader gets more familiar with this language. Second, we showed different Join languages that we represent by a single hypothetical language Polyphonic Scala. Since Polyphonic Scala and the later shown JEScala are both Scala based, programs implemented in each of these languages are the same, except for the part wheres JEScala is different from a Join language.

3.1 Case Study

Our case study is a Web server that hosts two applications: an online booking application for flight tickets `OB` and a marketplace application `MP` (Figure 3.1). This code uses the keyword `object` instead of `class`. In Scla this results in a singleton object as we explained in Section 2.2.1. In both applications, client requests are managed by `handle` methods (Lines 2 and 7). Since the Web server shares the host with other services, we want to ensure that all services are responsive in the presence of a high number of requests. To do so, when the load is high, we limit the number of concurrently handled requests by controlling the execution frequency of `handle` methods [1]. This requires to enforce a coordination schema among the threads executing the components of Figure 3.1.

Specifically, under high load, clients need to consume a token to be admitted into the server. When the load is high, client threads should block at the boundary of the `handle` method of each application, waiting for a token to be produced by the Token Generator (Line 12 in Figure 3.1).

Using regularly generated tokens that can be buffered in a small *bucket* is used for simple traffic shaping by using a token bucket filter [82] on IP networks. The buffered tokens from the bucket make short bursts possible,

3.2 Instrumentation

Figure 3.2 shows the instrumentation of the basic components of Figure 3.1, which the coordination logic (implemented by the object `CL` in Figure 3.3) uses to control the basic components.

The `beforeHandle` methods are called (Lines 3, 9) in the bodies of the `handle` methods to notify the coordination logic of the arrival of a new request to *one of* the applications. By calling a dedicated method of the coordination logic for each application, the coordination logic can be changed without requiring further changes to the web applications.

Similarly, the availability of new tokens is notified to the coordination logic by calling the `unblock` method each time a new token is about to be generated (see Line 15).

1. As in real Web servers, we assume that client connections not timely routed to applications are dropped after a timeout by the client or by the network

```
object OB { // OnlineBooking App
    def handle(...)= {
      ...
    }
}
object MP { // MarketPlace App
    def handle(...)= {
      ...
    }
}

object TG extends Thread { // Token Generator
    def createToken()= {
      ...
    }
  override def run = while(true) {
     sleep(1000)
     createToken()
  }
}

```

Figure 3.1 – Web Server: basic components.

3.3 Coordination Logic

The top level of the coordination logic is implemented by the object `CL` (Lines 2–16 of Figure 3.3), which turns the notifications from the basic components into calls to the object `RL` implementing rate limitation (Lines 19–33). Essentially, the calls from the web applications return immediately when the load is low, allowing the requests to be handled, which causes calls to the `block` method of the rate limiter otherwise. The calls to `unblock` are just forwarded to the rate limiter.

The role of the rate limiter is to delay returning from invocations to `block` until a token is available, that is, until an invocation to `unblock` occurs. We can represent the interaction between `block` and `unblock` calls as a state machine (Figure 3.4). When the rate limiter is in state `Free` (there is no application waiting for handling a request), calls to `unblock` are simply ignored, while calls to `block` switch the state to `Busy` (there is an application waiting). Then, a call to `unblock` brings back the rate limiter to state `Free`.

```
1  object OB {
2    def handle(...)= {
3      CL.OB_beforeHandle()
4       ...
5    }
6  }
7  object MP {
8    def handle(...)= {
9      CL.MP_beforeHandle()
10      ...
11   }
12 }
13 object TG extends Thread{
14   def createToken()= {
15     CL.unblock()
16     ...
17   }
18  ...
19 }
```

Figure 3.2 – Instrumented basic components.

We showed a Joins-based finite state machine in Section 2.5.3. Its implementation is based on a common technique in Join languages (see, for instance, [9, 51, 79]) by representing states as pending data receptions. We apply the same technique, with synchronous and asynchronous method calls, to the aforementioned state machine (see Figure 3.4). A simple invariant underlies the technique: there is always one single pending asynchronous method call in the object implementing the state machine. This pending call represents the current state. We refer to the corresponding methods (`free` and `busy` in our example) as *state methods*. The actions responsible for state transitions can be implemented by either synchronous or asynchronous method calls. We refer to these methods (`block` and `unblock`) as *action methods*. Each transition is implemented as a binary pattern between a state method and an action method (Lines 20, 24 and 28). The body executed when matching this pattern calls a state method. This maintains the invariant. The invariant is established when initializing the object by calling the state method for the initial state (`free`, Line 32). Note that a single pending call of a state method implies that the machine can handle only a single action at a time, the next action can only take place after the body has called a new state method.

Whereas the `unblock` method headers (Lines 24 and 28) are also declared as asyn-

```
1  // Coordination Logic
2  object CL {
3     def OB_beforeHandle()={
4         mayBlock()
5        }
6     def MP_beforeHandle()={
7         mayBlock()
8        }
9     def mayBlock()={
10        if (systemLoadHigh())
11          RL.block()
12     }
13    def unblock()={
14        RL.unblock()
15    }
16 }
17
18 // Rate Limiter as 2-state FSM
19 object RL {
20    def block():Unit & free():async {
21     // Stat.toBusy()  // future extension
22        busy() // free to busy
23    }
24    def unblock():async & busy():async {
25        // Stat.toFree() // future extension
26        free() // busy to free
27    }
28    def unblock():async & free():async {
29        // Stat.toFree() //future extension
30        free() // absorbed unblock
31     }
32    free() // initial state in constructor
33 }
```

Figure 3.3 – Coordination Logic with Polyphonic Scala.

chronous in order not to block the token generator, calls to `block` are synchronous. The Join pattern on Line 20 blocks the underlying thread until the state is free.

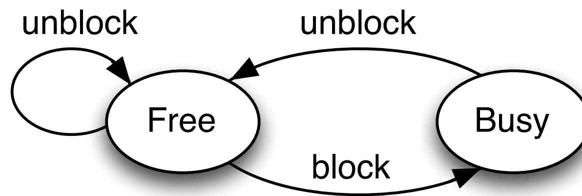


Figure 3.4 – Rate limitation as a state machine

3.4 Discussion

Even if the coordination schema is simple, its realization in Figure 3.3 has limitations.

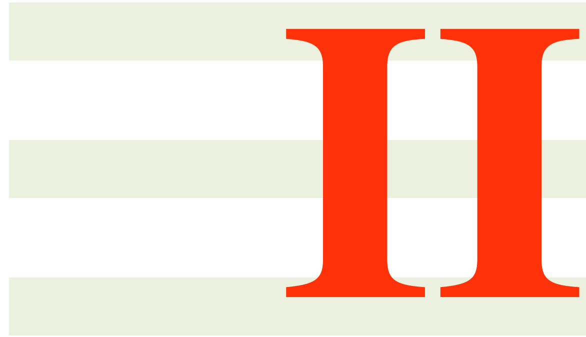
First, the components to be coordinated are intrusively modified to add the notifications (Lines 3, 9, and 15 of Figure 3.2). These modifications are necessary to make the points in the execution that are relevant to the coordination schema observable. Further modifications may have to be considered, both in the application components and the coordination logic for further extensions. For instance, we will consider in Section 4.3 adding a component `Stat`, which will require the modifications commented out on Lines 21, 25 and 29.

Second, several indirections are needed: one to deal with requests to *any of* the applications (Lines 3 and 6 of Figure 3.3) and another to take the load condition into account (Line 10). In the first case, the indirection implements a “union” semantics, which is not explicit in the code. Alternatively, the indirection can be suppressed by directly calling `mayBlock` within the applications. Yet, the cure is worse than the disease. In Figure 3.2, the applications call distinct `beforeHandle` methods and the union of these calls is properly modularized within the coordination schema. Calling `mayBlock` directly from the applications would move part of the coordination logic (a call from *either* `OB` *or* `MP`) away from the coordination schema, and hardcode it into the coordinated components. As a result, it would be, for instance, impossible to implement a balancing strategy between `OB` and `MP` just by modifying the coordination code.

To summarize, the logic of the coordination schema is hardly captured by the abstractions of a language with only abstractions from the Join calculus. Instead, it must be harvested from the calls inserted within the coordinated components in order to capture the events of interest and the logic of corresponding (possibly multi-header) bodies in the coordination code. In addition, the application is not extensible and requires invasive changes to introduce new components.

Our objective in this thesis is therefore to complement the abstractions provided by

the join calculus with means to untangle coordination schemas from base code, a separation of concerns reminiscent of Aspect-Oriented Programming and facilitating the development, understanding, and evolution of applications, with here a specific focus on concurrency.



Contribution

A Rich Event System to the Rescue

Contents

4.1 Basic Concepts of JEScala 91
- 4.1.1 Asynchronous events with Implicit Invocation 92
- 4.1.2 Imperative events 92
- 4.1.3 Implicit events 93
- 4.1.4 Declarative events 94
- 4.1.5 Abstract events 95
- 4.1.6 Joins on implicit and declarative events 95

4.2 Interacting disjunctions 97
- 4.2.1 Disjunctions consume and fire events 99
- 4.2.2 Multiple disjunctions inside the same class 99

4.3 Dynamic registration of handlers 101

4.4 Summary on events 103
- 4.4.1 Primitive events 103
- 4.4.2 Sequential event expressions 103
- 4.4.3 Join event expressions 104
- 4.4.4 Inheritance 104

The requirements for both being able to capture and combine program execution points are, of course, highly reminiscent of the join points and pointcuts of Aspect-Oriented Programming (AOP). AOP would indeed make it possible to capture in a pointcut, in a non-intrusive way, the fact that a request is about to take place. This also makes it possible to combine such pointcuts in order to deal with several kinds of requests under a specific condition, namely the fact that the load is high. Unsurprisingly, assuming the availability of AOP facilities within Polyphonic Scala would improve the implementation of our case study.

Still, some discrepancy would remain between the composition of join points via pointcuts using logical operators and the composition of channel endpoints via the join operator. Previous work on the languages ECaesarJ [88] and EScala [44] (Section 2.3.1), which did not provide any support for concurrency, suggests a way to eliminate this discrepancy. The main idea behind this previous work is that the join points of AOP and the explicit triggering of events encountered in event-driven programming are of the same nature. A join point can be seen as an occurrence of an *implicit event*, whereas an *explicit event* is explicitly triggered. Composite events can then be created by composing events through logical operators. The basic idea to solve our discrepancy issue is therefore twofold:

— Let us consider a data emission as triggering an explicit event.

— Let us consider the join operator as an additional composition operator.

Of course, this is not enough to get (implicit) concurrency: we also need a way to choose between synchronous and asynchronous event triggering.

All these ideas have been injected into EScala, leading to JEScala, described in detail in the next section. Using JEScala, our case study (Figure 3.3) can be rewritten as shown in Figure 4.4b without the limitations previously discussed. To help the reader compare the implementations, we repeat the first version in Figure 4.4a.

The term *event* can be confusing as, beyond the general idea that it refers to a noteworthy state change, the semantics of event constructs varies a lot. In particular, the events of ScalaJoins and the explicit events of (J)EScala share some characteristics: they are instance members and triggered using method call syntax. In both cases, events make it possible to split the traditional way that methods are defined. The *event* itself corresponds to a method header and its *event handler* corresponds to a method body. In ScalaJoins, these two parts have to be defined in the same class in order to be bound. This is quite different in (J)EScala where events and handlers can be defined in different classes and bound to each other in yet another class. There may also be different bindings, hence different handlers attached to the same event, resulting in some form

of implicit invocation [43]. Hence, unlike languages that model channels with functions/methods, data is not sent to a single destination but to multiple destinations, i.e., triggering an event corresponds to an emission on multiple channels.

Implicit invocation is central to our event model. It exchanges the rigid connection between a sender and its single receiver for a flexible broadcast mechanism. It is possible to change the group of receivers without modifying the sender. Implicit invocation makes no assumption on the number of receivers. Therefore, it does not require to create dependencies that are not actually needed (e.g. a tracer can be detached). Receivers can be added and removed at runtime.

4.1 Basic Concepts of JEScala

We presented in Section 2.5.2 an overview of Join languages. All Join languages that we discussed offer synchronous and asynchronous channels. When code sends data to an *asynchronous* channel, its execution continues without waiting. But the execution of code that sends to a synchronous channel waits for a confirmation from a receiver. In some Join language is a return value the confirmation that resumes the execution of a sender.

The channels of a Join language have only a single receiver, which creates that channel. The Object-Oriented Join languages like Polyphonic C# and Join Java model channels as methods. These methods are a part of the receiving object.

We discussed event systems in Section 2.2, a key element of these systems is implicit invocation. Unlike channels in Join languages, event systems do not constrain the number of receivers for an event. Dynamic registration in event systems makes it possible for a receiver to register with an event. This registration can happen added after the event was defined, without any preparation for the number of handlers that observe the event.

We extend the EScala event system, which we explained in Section 2.3.1. Events in EScala are source constructs. Triggering an event results into an event *occurrence*. Since this distinction is in most cases clear, we often use event as a shorter form for event occurrence. Defining the type of the parameters is part of defining an event, the observers of that event need to be able to deal with parameters of that type. After we explained the EScala event system that we enriched with asynchronous events, we explain the integration of join patterns that combine into explicit disjunctions.

event-decl ::= *prim-event* | *decl-event*
prim-event ::= **imperative** [*sync-modifier*] **evt** *event-name*
decl-event ::= **evt** *event-name* **=** *event-express*
  | **evt** **(***event-name* { **,** *event-name*}$^{+}$ **)** =
  (*join-express* { **|** *join-express*}$^{+}$ **)**
event-express ::= [*obj-ref* **.**] *event-name*
  | **super** **.** *event-name*
  | *event-prefix-operator event-express*
  | *event-express event-infix-operator event-express*
  | *event-express fun-operator fun*
  | *implicit-event*
implicit-event ::= [*obj-ref* **.**] *implicit-selector* **(** *method-name* **)**
implicit-selector ::= **beforeSync** | **afterSync**
  | **beforeAsync** | **afterAsync**
  | **before** | **after**
sync-modifier ::= **sync** | **async**

event-prefix-operator ::= **!!**
event-infix-operator ::= **||** | **&** | ...
fun-operator ::= **map** | **&&** | ...
join-express ::= **(** *event-name* { **&** *event-name* }$^{+}$ **)**

Figure 4.1 – The Syntax of JEScala.

4.1.1 Asynchronous events with Implicit Invocation

We present the event system of JEScala. For reference, the syntax of JEScala that is relevant for the discussion is given in Figure 4.1. JEScala inherits the event system of EScala, while extending it with asynchronous execution semantics. Yet, the extension is designed to ensure backward compatibility with EScala: A program that does not use new features introduced by JEScala is still a valid EScala program.

4.1.2 Imperative events

The explicitly-triggered events, called *imperative* in EScala, are a language construct for implicit invocation, that we also find in other languages (for example the delegates in C#).

In JEScala, the declaration of each explicitly triggered event begins with the keyword

`imperative`, followed by an optional `sync` or `async` modifier. Hence, the programmer specifies what happens after triggering an imperative event. Triggering a synchronous event implies waiting until all its associated handlers return. The alternative is to trigger an asynchronous event. The handlers of an asynchronous event are executed concurrently with the code that triggered the event, without blocking the execution of the code that triggers an asynchronous event. By including synchronicity into the interface of the entity that declares the event, it is possible to express whether the progress of code that triggers an event can temporary block.

The execution of code that triggers a synchronous event waits for all its associated handlers to return. However, this does not imply that the event is completely handled, since the handlers can spawn new computations via mechanisms as Scala futures or a handler can trigger new asynchronous events.

When the synchronicity of an event is not explicitly defined, using synchronous as default ensures that EScala programs keep their original semantics [44].

4.1.3 Implicit events

An alternative to explicitly triggering events in EScala is using implicitly triggered events. These are triggered when the program flow reaches a referable point in the execution. In Aspect-Oriented languages, such points are known as join points.

In EScala it is possible to mark methods as observable, which makes implicit events available that signal the begin and the end of their execution. For observable methods, EScala provides the implicit events `before(`*method-name*`)` and `after(`*method-name*`)` that are executed when a method enters – respectively, finishes – its execution. While in most AOP languages all methods are observable, methods in EScala are only observable by the same object unless a methods is marked as such in their interface. This does not only control the exposure of the methods, it also helps the compiler to decide which methods need to be instrument. We extended EScala's implicit events to account for synchronicity. In JEScala four implicit events are available: `beforeSync`, `beforeAsync`, `afterSync` and `afterAsync`. They all take a method name as argument. In the spirit of remaining compatible with EScala, the events `before(`*method-name*`)` and `after(`*method-name*`)` are still valid and mapped to the corresponding synchronous events.

4.1.4 Declarative events

Similarly to EScala, JEScala also supports declarative events defined in terms of event expressions. The event expressions are defined by composing other events through operators. The most important operators are union (`||`), filter (`&&`) and transform (`map`). Using these operations results in Event expressions like in e_1`||`e_2 (occurrence of one among e_1 or e_2), e_1`&&`p (e_1 occurs and the predicate p is satisfied). The filter operator is overloaded with the logical and operator as in EScala. We did keep the `&&` operator to remain compatible with EScala, although we would prefer the word `filter` as an alternative, since that is similar to collections in Scala. Declarative events have no synchronicity by themselves. Instead, they inherit the synchronicity of the event that triggers them. For example, the event e_1`&&`p is executed synchronously – respectively asynchronously – if e_1 is synchronous – respectively asynchronous. Note that there is no ambiguity, since the `||` composite event is triggered by only one event and inherits its synchronicity from that event.

```
1 imperative async evt e1
2 sync evt e2 = e1 && (predicate)
```

Figure 4.2 – Conflicting synchronicities for declarative events.

This design choice requires more explanations. Providing a `sync`/`async` modifier for declarative events makes it possible to express event combinations like in Figure 4.2, whose semantics is unclear. The imperative event `e1` is asynchronous, so one expects that all the handlers bound (even indirectly) to the event are executed asynchronously. However `e2`, which depends on `e1`, is declared synchronous. This should imply that handlers attached to `e2` are executed immediately, which contradicts the modifier of `e1`. For this reason we decided to avoid explicit synchronicity for declarative events and let them inherit the synchronicity of the event that triggers them.

```
1 imperative sync evt e1[Int]  // triggered input event
2 imperative async evt e2[Int] // for  handlers
3 e1 += ((arg:Int) => e2(arg) )
```

Figure 4.3 – Simulation of evt e2 = !! e1.

It still makes sense to force a declarative event to be asynchronous. To this aim, the prefix `!!` operator converts a possibly synchronous event expression into an asynchronous one. Figure 4.3 shows an alternative implementation of the expression `evt e2= !!e1` that defines an imperative asynchronous event `e2` (Line 2) with the same type of argument as `e1` (Line 1). A handler (Line 3) for `e1` explicitly triggers the imperative event `e2`

with the same parameter. Our simulated version does not prevent the programmer from explicitly triggering the `e2` event, which is only a shortcut for triggering `e1`. Declaring `e2` in our simulation as private, would require additional code to register handlers with the event.

The `!!` operator has no counterpart to change an asynchronous event into a synchronous event, since an event handler registered with an asynchronous event cannot make the code wait after triggering the event.

4.1.5 Abstract events

In EScala, it is possible to declare *abstract* events that are defined in concrete subclasses. In JEScala, the synchronicity of abstract events is not specified.

This design decision is motivated by the fact that an abstract event can be overridden by either a primitive or a composite event. Allowing synchronicity modifiers in abstract events would, for instance, allow one to define an abstract `sync` event and override it in a subclass with a declarative event – running into the trouble previously described.

Actually, the synchronicity of an abstract event cannot be known until it has been defined. Defined as a primitive event, its synchronicity is known statically. Defined as a composite event, its synchronicity may not be known until runtime, depending on its definition and its evaluation.

4.1.6 Joins on implicit and declarative events

As already mentioned, the key feature of JEScala is the combination of the rich event system described above with join concepts. This combination enables a succinct and well-localized definition of synchronization logic. We illustrate this by comparing the Polyphonic Scala implementation that we repeated as Figure 4.4a with the JEScala code in Figure 4.4b, which shows the implementation of the Web server example by using Join patterns with implicit and declarative events.

```
// Coordination Logic
object CL {
  def OB_beforeHandle()={
      mayBlock()
  }
  def MP_beforeHandle()={
      mayBlock()
  }
  def mayBlock()={
    if (systemLoadHigh())
      RL.block()
  }
  def unblock()={
     RL.unblock()
  }
}






// Rate Limiter as 2-state FSM
object RL {
  def block():Unit & free():async {
    // Stat.toBusy()  // future extension
    busy() // free to busy
  }
  def unblock():async & busy():async {
    // Stat.toFree() // future extension
    free() // busy to free
  }
  def unblock():async & free():async {
    // Stat.toFree() //future extension
    free() // absorbed unblock
  }
  free() // initial state in constructor
}
```

(a)

```
// Coordination Logic
object CL {
  evt block =
          (OB.beforeSync(handle) ||
            MP.beforeSync(handle) ) &&
            systemLoadHigh()
  evt unblock =
          TG.beforeAsync(createToken)
}

// Rate Limiter as a  Free-Busy FSM
object RL {
  imperative async evt free[Unit]
  imperative async evt busy[Unit]
  evt (toBusy, freed, absorbed) =
     ( CL.block  & free
     | CL.unblock & busy
     | CL.unblock & free  )
  evt toFree = freed || absorbed
  toBusy += ((arg:Any)=>busy())
  toFree += ((arg:Any)=>free())
  free() // initial state
}
```

(b)

Figure 4.4 – Coordination Logic with Polyphonic Scala (a) and JEScala (b).

The execution of `handle` in `OB` and `MP` is captured by the implicit events `OB.beforeSync(handle)` and `MP.beforeSync(handle)`, which are composed so that the declarative event `block` (Line 5) only fires when the load is high. The implicit event `TG.beforeAsync(createToken)`, which captures the creation of a token is aliased to `unblock` (Line 8).

Finally, the state machine from Figure 3.4 is implemented as described in Chapter 3, except that state and action methods are replaced by *state* and *action events* respectively: `free` and `busy` (declared Lines 13 and 14), and `block` and `unblock` (defined in `CL` Lines 5 and 8). The implicit disjunction of Figure 4.4a is replaced by an explicit one (Line 15). Each alternative combines a state and an action event and triggers one of the events `toBusy`, `freed` or `absorbed` when it matches. These new events are necessary to attach a handler to each alternative and have the advantage that sharing can be made explicit, here by defining the event `toFree` that signals a transition to the `Free` state. Triggering the `free` event (Line 22) sets the initial state of the machine.

The JEScala implementation in Figure 4.4b has several design advantages compared to Figure 4.4a, which uses Polyphonic C#, our synthetic Join Language. The coordination logic is captured in one place (lines 5 – 23). There is no footprint of the coordination logic in the components to coordinate; all relevant execution points for the coordination are captured by implicit events. As a result, the coordination logic is properly modularized. The coordination schema is expressed declaratively, improving clarity and extensibility, e.g., the balancing strategy is clearly captured at Line 5 thanks to event expressions. As a result, introducing an additional application in the coordination schema is e.g., as straightforward as adding a new implicit event in Line 5. Given that events are values, the coordination schema can be a separate reusable component parameterized by events to coordinate. For the sake of simplicity, we have used singleton classes in our example. A more realistic implementation of the rate limiter would use a class whose primary constructor would take a `block` and an `unblock` event as parameters.

4.2 Interacting disjunctions

To introduce more abstractions of JEScala we extend the Web server example. So far, there is no distinction in serving `OB` and `MP`. As a result, a high request rate in one application can significantly slow down the other. To address this issue, we shall introduce load distribution between `OB` and `MP`. Furthermore, we shall improve the token generation to avoid that tokens accumulate when they are generated at a higher frequency than requests from clients arrive – in the new version of the Web server, unused tokens simply expire after some time. The implementation of the extended Web server is shown in Figure 4.5. Instead of immediately discarding surplus `unblock` events by a state ma-

```
1  object CL {
2    evt blockOB = OB.beforeSync(handle)
3    evt blockMP = MP.beforeSync(handle)
4    evt unblock = TG.beforeAsync(createToken) map (()=>currentTime)
5  }
6  object RL {
7    imperative sync evt requestUnblockOB[Unit]
8    imperative sync evt requestUnblockMP[Unit]
9    imperative sync evt innerBlockOB[Unit]
10   imperative sync evt innerBlockMP[Unit]
11
12   CL.blockOB += (()=>{ requestUnblockOB(); innerBlockOB() })
13   CL.blockMP += (()=>{ requestUnblockMP(); innerBlockMP() })
14
15   evt _ = innerBlockOB & grantUnblockOB
16   evt _ = innerBlockMP & grantUnblockMP
17
18   evt (mayUnblockOB, mayUnblockMP) =
19       (requestUnblockOB & CL.unblock) |
20       (requestUnblockMP & CL.unblock)
21
22   evt grantUnblockOB = mayUnblockOB   && ((ts)=> !isExpired(ts))
23   evt expiredUnblockOB = mayUnblockOB && ((ts)=> isExpired(ts))
24   expiredUnblockOB += (()=>requestUnblockOB())
25
26   evt grantUnblockMP = mayUnblockMP   && ((ts)=> !isExpired(ts))
27   evt expiredUnblockMP = mayUnblockMP && ((ts)=> isExpired(ts))
28   expiredUnblockMP += (()=>requestUnblockMP() )
29 }
```

Figure 4.5 – Distributing load among Web Applications in JEScala.

chine (Figure 3.4), this extension accumulates `unblock` events, which are consumed without effect after they expire. Since the example is not trivial, we start with a high-level description of the coordination schema. When the client requests arrive, which is exposed by events in Lines 2–3, each client is blocked until unblocking is granted (Lines 15–16). If two requests from different applications are performed, only one is chosen non-deterministically (Lines 18–20). The authorization to proceed is given only by tokens that are not expired. The expiration for tokens is implemented in Lines 22–28.

For a detailed description of the coordination schema of Figure 4.5, consider the flow of the events associated to `OB` (the event flow for `MP` is similar). When a client request for `OB` arrives, `blockOB` is synchronously triggered (Line 2). Its handler (Line 12) triggers `requestUnblockOB` and is blocked in the disjunction Line 19, waiting for an `unblock` event. The `unblock` events are generated by attaching a timestamp to events produced by the token generator (Line 4). The selection of a pattern in the disjunction pattern triggers a `mayUnblockOB` event, its handler proceeds but blocks at once on the triggering of the `innerBlockOB` event, involved in the disjunction Line 18. Concurrently, depending on whether the token is expired or not, either a `grantUnblockOB` event is triggered (Line 22) releasing the `blockOB` handler waiting at the disjunction Line 15 or a `requestUnblockOB` event is regenerated (Line 24) and the handler remains blocked at the disjunction (Line 19). In the first case, the `blockOB` handler returns at once and the application can proceed.

The example in Figure 4.5 demonstrates several aspects of the design of JEScala.

4.2.1 Disjunctions consume and fire events

Like in other Join languages, disjunctions can be used to compose multiple conceptually-related Join patterns; multiple Join patterns in a disjunction can share an event (see e.g., Lines 15–16), whereby, the first that matches consumes the shared event. If multiple patterns match, the one that fires is chosen non-deterministically. In JEScala, *disjunctions fire events*; it is possible to distinguish the join that fires by associating a specific event to each join in the disjunction (see e.g., Line 18). In other Join languages when a Join pattern matches, a handler is executed. The model of JEScala is homogeneous: events generated by a disjunction can be freely composed with other events. For example, a union operator `||` can be used to merge the events from the same disjunction, if we do not need to distinguish among them. The same effect can be achieved in other Join languages only by triggering the same emission from each handler registered to a Join pattern, which unnecessarily resorts to imperative code and bloats the coordination logic.

4.2.2 Multiple disjunctions inside the same class

The example in Figure 4.5 shows another feature of JEScala. Many Join languages group all Join patterns defined in an object into a single *implicit* disjunction associated to the object. In JEScala, each disjunction *explicitly* defines a set of Join patterns that are checked for a possible match. Therefore, JEScala allows one to define *multiple disjunctions inside the same class*. This fosters modularity. In case of complex coordination

schemas, like in Figure 4.5, several disjunctions are required. Some of these disjunctions (Lines 15 and 16) are reduced to a single pattern, which represents a alternative.

In languages with implicit disjunctions, we would need both to create an artificial disjunction with two alternatives composing the two disjunctions with a single alternative (if we want to keep them in the same class) and split the coordination patterns into at least two separate classes in order to also implement the second disjunction (Line 18). In JEScala, disjunctions that logically belong to the same schema can be properly modularized inside the same class.

```
1  class CorrectJoinClass {
2    imperative sync   evt a[Unit]
3    imperative sync   evt b[Unit]
4    imperative async  evt c[Unit]
5    imperative async  evt d[Unit]
6    evt (j1,j2)= b & c  |
7                 a & d
8    j1 += ((arg:Any) => d())
9    j2 += ((arg:Any) => c())
10   b()  // trigger in constructor
11 }
12 class BrokenJoinClass extends CorrectJoinClass {
13   ..
14   imperative async evt e[Unit] // additional event
15   evt (j1,j2,j3)=  b & c  |
16                    a & d     |
17                    a & d & e
18   ..  // no additional handlers
19   e() // additionally trigger in constructor
20 }
```

Figure 4.6 – Additional patterns cause a deadlock.

Supporting multiple disjunctions also affects the interaction between join abstractions and Object-Oriented inheritance. Figure 4.6 shows a superclass (`CorrectJoinClass`) that defines a disjunction with two Join patterns among events a, b, c, and d (Lines 6 and 7). This example alternates its consumption of events between the synchronous events `a` and `b`. The synchronous events `a` and `b` are triggered by code that we omitted. We add a Join pattern with events a, d and the additional `e` (Line 14) in a subclass (`BrokenJoinClass`). We illustrate this in the code by replacing the disjunction by one with a additional pattern (Line 7). Because the event `e` is triggered by the constructor

of the subclass, the additional pattern (Line 17) can match instead of the second pattern (Line 16). Then the waiting `d` event is consumed and the senders of the synchronous events `a` and `b` remain blocked, unless an event `c` or `d` is triggered. The same problem can appear with implementing an FSM with patterns that each consume a *state* event and an *action* event. when after consuming a waiting *state* event no other *event* is triggered the FSM would dead lock as well. This problem is described by [9] for Polyphonic C#, which uses objects as implicit disjunctions. To tackle these issues, Object-Oriented Join languages impose limitations on inheritance to prevent adding new Join patterns in subclasses. In Join Java, only final classes can define a join; in Polyphonic C# it is possible to override an inherited disjunction by replacing the associated handler, but it is not possible to add a join. Since in JEScala Joins defined within a class are not implicitly correlated into a disjunction, classes can be freely extended regardless of the presence of disjunctions. Yet, advised by the lesson learned from existing Join languages, we forbid breaking existing disjunctions by extending them with new Join patterns. In our design, subclasses can only entirely override them, by defining a new disjunction that overrides the result events that are defined by a disjunction in the superclass.

4.3 Dynamic registration of handlers

While handlers are statically bound to Join patterns in existing Join languages, the handlers in JEScala can be dynamically (un)registered. Figure 4.7 shows an extension of the Web server from Figure 4.4b that computes statistics about the queuing time of the applications inside the server. Logging the time spent by the rate limiter (RL) in the busy state requires the observation of entering the states `Free` and `Busy` in the rate limiter. We prepared the code in Figure 4.4a by inserting explicit method calls into the `RL` component (Lines 26, 30 and 34). In JEScala, we just need to register additional handlers (Line 16) to the exposed events `toFree` and `toBusy` of RL without modifying the code of RL(Figure 4.4b). By using the declarative event `toFree` we can register each of our handlers with a single event with a descriptive name.

By triggering the `enable` event the handlers `toBusy` and `toFree` (Figure 4.7) are registered. Since we are not all the time interested in these statistics, triggering the `disable` event removes the handlers. The internal declarative events `doEnable` and `doDisable` prevent incorrect double registrations. The anonymous handler of `doEnable` registers the handlers from `Stat` with the exposed events from RL. Its counterpart for `doDisable` unregisters these handlers again. The handler sets the `isEnabled` flag that is used by the filters defining `doEnable` and `doDisable`. An implementation with Polyphonic Scala (the language also used in Figure 4.4a), without declarative events needs to integrate conditions into the methods `toBusy` and `toFree` to enable and dis-

```
1  object Stat {  // Statistics
2    var sTime:timeStamp=null
3    def hdlToBusy:Unit= { // handler
4      sTime=currentTime          }
5    def hdlToFree:Unit=if (sTime!=null) {
6      log_busy(currentTime-sTime)
7      sTime=null                 }
8    var isEnabled: Boolean = false
9    imperative sync evt enable[Unit]  // trigger to enable
10   imperative sync evt disable[Unit] // trigger to disable
11   evt doEnable = enable && (()=>!isEnabled ) // enable only once
12   evt doDisable = disable && (()=>isEnabled )
13   // register an anonymous handler with doEnable event
14   doEnable  += (()=>{
15     isEnable=true
16     RL.toBusy += hdlToBusy _ // register with RL.toBusy event
17     RL.toFree += hdlToFree _   } )
18   doDisable += (()=>{
19     RL.toBusy -= hdlToBusy _ // unregister
20     RL.toFree -= hdlToFree _
21     isEnabled = false          } )
22 }
```

Figure 4.7 – Dynamic handler registration in JEScala.

able the functionality at runtime.

In other Join languages, dynamic binding between Join patterns and handlers can be obtained only by adding a layer of indirection with an intermediate handler that is responsible for notifying the right handler in case a certain condition is met. This approach has the drawback of moving the event logic from high-level operations among events to handlers. Further, it introduces a performance penalty, because the intermediate handler is *always* notified regardless of whether a reaction is needed. More importantly, this solution does not properly handle situations where the binding depends on the execution. For example, in an extensible game could a player implement extensions by using handlers, and third party vendors can add figures that emit events. JEScala solves these issues thanks to the uniform representation of Join pattern outputs as events and dynamic event handler registration.

4.4 Summary on events

We show examples of how events are defined in the code and what effect this has on synchronicity.

4.4.1 Primitive events

The way to trigger primitive events divides them into two groups: the explicitly triggered imperative events and the implicitly triggered events that are not defined by the programmer but bound to a name by a declarative event. Figure 4.8 shows first two events that are triggered explicitly and then it shows how to access two implicit events that mark the begin or the end of the execution of the `myMth` method. The events bound by`beforeSync` and its asynchronous counterpart have a tuple parameter with the arguments of each method call. The events bound via an `afterSync` or `afterAsync` have a argument that is a pair, the first part is a tuple composed of the argument, the second part is the return value of the method call.

Code	Description
`imperative sync evt synEvt[Unit]`	synchronous event without argument
`imperative async evt asynEvt[Int]`	asynchronous event with an `Int` argument
`evt waitingToStart=beforeSync(myMth)`	triggered before myMth (sync)
`evt done=afterAsync(myMth)`	triggered after myMth (async)

Figure 4.8 – Examples of primitive events.

4.4.2 Sequential event expressions

These operators are inherited from EScala. However, JEScala supports both synchronous and asynchronous events. Event expression can describe a union (`||`) event occurrences from two events. They can transform the argument of an event by applying a function on each event occurrence with the `map` operator. Event expressions can also filter event occurrences by applying a predicate function on each event occurrence. Except for the double bang (`!!`) no operator changes the synchronicity of the event occurrences. Because event expressions do not changes the synchronicity of event occurrences, the occurrences of a synchronous and an asynchronous event keep their original synchronicity. Figure 4.9 shows the synchronicity of event expressions based on `sEv[Int]`, a synchronous event and `aEv[Int]` an asynchronous event. The events defined in this

table begin with a letter that indicates synchronicity. The *triggered* column show from top to bottom a sequence of events the *result* shows after which event an event emits a resulting event occurrence.

Expression	Triggered	Result
`evt varEv=aEv || sEv`	`aEv(1)` `sEv(2)`	varEv(1): async varEv(2): sync
`evt aDouble=aEv map (x=>x*2)`	`aEv(2)`	aDouble(2)
`evt sDouble=sEv map (x=>x*2)`	`aEv(2)`	sDouble(2)
`evt aOdd=aEv && (x=>(x%2!=0))`	`aEv(1)` `aEv(2)`	aOdd(1) -
`evt sEven=sEv && (x=>(x%2==0))`	`sEv(1)` `sEv(2)`	- sEven(2)

Figure 4.9 – Examples of sequential event expressions.

4.4.3 Join event expressions

In JEScala, events can have varying synchronicity, this also affects Join patterns. The synchronicity of the resulting event of a matching Join pattern is only synchronous when at least one synchronous event occurrence was involved. Disjunctions combine Join patterns and define a tuple of events, for each Join pattern in a disjunction is a resulting event in the resulting tuple. Figure 4.10 shows disjunctions with patterns of events (e_1, e_2, ...). To show the varying synchronicity in this figure, each event e_x is a union of the synchronous event sE_x and the asynchronous event aE_x. In order to save space, none of these events has an argument. The *triggered* column shows from top to bottom for each expression a sequence of events, the *result* shows the resulting event after the triggered event that caused it.

4.4.4 Inheritance

Similarly to EScala, JEScala considers events as object members that can be overridden in a subclass. Important is that overriding a concrete event cannot change the type of the argument. Abstract classes can define abstract events. In that case a subclass can define an event expression about how the abstract event is triggered. In the design of JEScala can only primitive events define the synchronicity of the event occurrences that they emit. Therefore declarative events cannot be defined with a specific synchronicity.

Expression	Triggered	Result
`evt ej1=e1 & e2`	$aE_1()$	
	$aE_2()$	ej1: async
`evt ej1=e1 & e2`	$sE_1()$	
	$aE_2()$	ej1: sync
`evt (ej1,ej2)= e1 & e2`	$sE_1()$	
`| e1 & e3`	$aE_2()$	ej1: sync

Figure 4.10 – Examples of Join expressions and disjunctions.

Indirectly this also prevents that synchronicity could be defined for abstract events, since it would mean that an abstract event specifies the synchronicity of a concrete declarative event that is defined in a subclass.

State Machines

Contents

5.1 State Machines using Object-Oriented abstractions 108
5.2 State Machines using Actors 109
5.3 State Machines using events and Join Patterns 111
5.3.1 Using events 111
5.3.2 Coordination using a State Machine 112
5.3.3 Using events and Join patterns 113
5.3.4 One step further 114
5.4 A DSL to describe an FSM 115
5.4.1 FSM definition 116
5.4.2 DSL design 118
5.5 Advanced State Machines 119
5.5.1 Action events with arguments 120
5.5.2 State events with an argument 124
5.6 Conclusion 125

The JEScala event system can express the coordination among concurrent components. Finite State Machines provide high-level general purpose means to express coordination

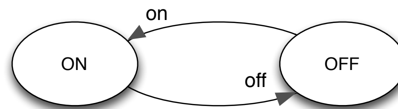


Figure 5.1 – On/Off switch as an FSM.

schemas. Our case study also included a Finite State Machine (FSM). We study them further in this chapter. First, we describe the State pattern [42] in a sequential setting, then we discuss FSMs that are driven by actions, which can occur concurrently. We show deterministic Joins-based state machines that are able to handle concurrent *action* events. Then we present a Domain Specific Language (DSL) for describing an FSM. The initial goal of the DSL is to provide the programmer with a standard description for an FSM that also reduces the amount of boilerplate code. We also use it to experiment with alternative implementations for a FSM. We extend the DSL to generate FSMs that use events without arguments. We also give an overview of state machines that use arguments for action events. We conclude by showing state machines that represent different states by using a state event with an argument.

5.1 State Machines using Object-Oriented abstractions

First we consider a solution using only the abstractions available in Object-Oriented languages. Figure 5.2 shows the implementation of a simple switch (Figure 5.1) using the State [42] design pattern. The switch has two states `ON` and `OFF` and two actions turn `on` and turn `off`, switching to state `ON` and `OFF`, respectively. The class `Switch` contains a trait `State` (Line 3) that declares the messages that a switch object can receive. The two inner singleton classes `ON` (Line 7) and `OFF` (Line 11) represent the two possible states of the switch behavior by implementing `State`. The current state of the switch (variable `currentState`, Line 2) is an object of type `State`. All state-dependent methods of the switch are forwarded (Lines 15, 16) to the current state. In a sequential setting, receiving a message in an inappropriate state must be handled in some way. We used a simple but abrupt policy here: an error is raised.

In a concurrent setting, a standard policy (but it is not the only one, see [72]) is for a caller to wait until its action can be applied. Handling concurrent action events can lead to race conditions since transitions can execute functions that modify shared data. To avoid race conditions, this requires a monitor, as shown in Figure 5.3. The switch methods (Lines 28, 31) are `synchronized` in order to grab the mutual exclusion lock associated with the monitor. Calls to `wait` (Line 10) and `notify` (Lines 17, 23) exclude

```
1  class Switch {
2    private var currentState:State = OFF
3    private trait State {
4     def on: Unit
5     def off: Unit
6    }
7    private object ON extends State {
8      def on = throw new IllegalStateException
9      def off = currentState = OFF
10   }
11   private object OFF extends State {
12     def on = currentState =  ON
13     def off = throw new IllegalStateException
14   }
15   def on = currentState.on
16   def off = currentState.off
17 }
```

Figure 5.2 – Sequential switch with the State pattern.

threads and permit them to reenter the monitor. The variable `outer` (Line 2) is defined as an alias to `this` so that notifications in nested objects are associated to the proper monitor.

5.2 State Machines using Actors

Java monitors are inherited by Scala. However, programming with Java monitors can be error prone. It is, for instance, very easy to forget a call to `notify`, which will not be noticed by the compiler and results in threads getting stuck. In Scala, an alternative is to use Actors. As shown in Figure 5.4, the actions are not implemented by methods any longer but by actor messages (Lines 2, 3). These messages are delivered in the actor's mailbox and handled through guarded pattern matching (Lines 13, 14). Message handling is asynchronous, since a client can proceed as soon as a message has been delivered (it could also be made synchronous by expecting a reply from the actor). This solution greatly improves over the thread-and-lock approach presented in the last paragraph. However, it is still unsatisfying from a modularization standpoint, since the entity that sends the message has to know in advance the receiving actor that implements the switch. Events and implicit invocation – as we will explain shortly – solve this issue

```
1  class Switch {
2    outer =>
3
4    private var currentState:State = OFF
5    private trait State {
6      def on: Unit
7      def off: Unit
8    }
9    private def await(cond: => Boolean) = while (!cond) {
10     wait()
11     }
12
13   private object ON extends State {
14     def on =  await (currentState == OFF)
15     def off = {
16       currentState = OFF
17       outer.notify()
18       }
19   }
20   private object OFF extends State
21     def on = {
22       currentState = ON
23       outer.notify()
24       }
25     def off = await (currentState == ON)
26   }
27   // actions
28   def on = synchronized {
29     currentState.on
30     }
31   def off = synchronized {
32     currentState.off
33     }
34 }
```

Figure 5.3 – Concurrent switch with the State pattern and a monitor.

and enhance decoupling.

```
1 // actor messages
2 case object On
3 case object Off
4
5 class Switch extends Actor {
6   private trait State
7   private case object ON extends State
8   private case object OFF extends State
9
10  private var currentState: State = OFF
11
12  def act():Unit = {  loop { react {
13            case On if currentState == OFF => currentState = ON
14            case Off if currentState == ON => currentState = OFF
15                  }       }       }
16 }
```

Figure 5.4 – Concurrent switch implementation using actors.

5.3 State Machines using events and Join Patterns

5.3.1 Using events

An alternative to implementing actions as actor messages is to implement them as events. An essential difference is the use of *implicit invocation*. One can think about the Observer pattern [42] or a publish/subscribe approach [36]. Whereas an actor message is sent to a specific actor (or a method call to a specific object), an event is triggered and *implicitly* received by the entities interested in the event. These entities can be modeled as objects in an Object-Oriented setting. The reception of an event results in the execution of the event handlers attached to the event. In the example below we illustrate how events decouple the entity triggering from the entities handling it.

Figure 5.5 shows a variant of the implementation with a monitor using events in JEScala. The states are handled as in Figure 5.3 but the actions `on` (Line 3) and `off` (Line 4) are now implemented as *action* events. These events do not carry any data, hence the explicit use of the type `Unit`. They are bound to handlers, with parameters of type `Unit` (lines 6 and 7). The handlers forward event handling to the state objects. With the events declared as *asynchronous* through the keyword `async`, the handlers are executed concurrently with the thread that triggers the events. It is also necessary to prevent the concurrent execution of handlers within the switch. Because of this, the body of each

```
1 class Switch {
2   outer =>
3   imperative async evt on[Unit]
4   imperative async evt off[Unit]
5   ...
6    on  += ((_) => synchronized { currentState.on  } )
7    off += ((_) => synchronized { currentState.off } )
8 }
```

Figure 5.5 – Concurrent switch with State pattern, monitor and events.

handlers is wrapped in `synchronized` that used the `Switch` object as a monitor. From the perspective of the client, this implementation is similar to the one using an actor, since once the event has been triggered, the client thread may proceed. Note, however, that this potentially requires more resources from the runtime than the solution using an actor. Indeed, compared to delivering a message to an actor, triggering an asynchronous event requires an additional thread. If many asynchronous action events are triggered concurrently, the same number of threads are needed. However, most of the threads could be blocked while waiting to gain access to the monitor.

But before explaining how this issue can be circumvented, let us first show the benefit of using events with respect to coordination.

5.3.2 Coordination using a State Machine

This benefit relies on two features available in JEScala: the availability of both *asynchronous* and *synchronous* events, and the support for both *explicit* and *implicit* events.

Combining both synchronous and asynchronous events When a *synchronous* event is triggered, the execution proceeds with event handling. This makes it possible, in a very lightweight manner, to decide whether event triggering should block a client or not. Let us for instance, consider the switch of Figure 5.5 and turn the action event `on` into a synchronous event by replacing the `async` modifier with the `sync` modifier. Such a switch can then be used as a concurrent lock. It can regulate access to a shared resource, that is implemented as a simple passive object without any specific synchronization code. Before accessing the critical section that must be protected an event `on` has to be emitted, after leaving it an event `off` has to be emitted. Here, a benefit of implicit invocation is that these events are handled by the switch but they could as well be handled by other objects, for instance a logger recording to the resource.

Using implicit events So far, we have assumed the use of *explicit* events: such events have to be *explicitly* triggered at the right places, for instance, in the code of our shared resource. An alternative is to use *implicit* events, which indicates to the compiler (or the events library) where events should be triggered in the *compiled code* (or at runtime) without cluttering programs with explicit event triggering. Implicit events are directly related to join points in Aspect-Oriented Programming [68].

This can be combined with class parametrization. Let us consider again the example of Figure 5.5. Instead of defining the action events as members of the class, they can be defined as parameters of the constructor: `class Switch(evt on[Unit], evt off[Unit]) {...}`. It is then possible to synchronize our previous resource, accessed through a call `r.use()`, by instantiating a switch as follows: `new Switch(r.beforeSync(r.use), r.afterAsync after(use))`, where, for instance, `r.beforeSync(r.use)` denotes the emission of a synchronous event just before the execution of the method `use` by the object `r`.

5.3.3 Using events and Join patterns

```
1  class Switch(evt on[Unit], evt off[Unit]) {
2    // state events
3    private imperative async evt ON[Unit]
4    private imperative async evt OFF[Unit]
5    private evt  (toggleON, toggleOFF) = (OFF & on) |
6                                         (ON & off)
7    toggleON  += ((_) => ON() )
8    toggleOFF += ((_) => OFF() )
9
10   OFF() // initialization to state OFF
11 }
```

Figure 5.6 – Concurrent switch with events and Join patterns.

Figure 5.6 shows a new variant of the switch that replaces the use of a monitor by the use of *Join patterns* and *disjunctions* thereof.

In Section 2.5.3 we showed a finite state machine that is based on a disjunction of Join patterns using the synthetic language based on Polyphonic C#. In JEScala the declarative event expressions with the union operator `||` combine incoming transitions as we see in Section 5.3.4. JEScala offers an additional feature compared to the other

discussed Join languages, by supporting both synchronous and asynchronous event occurrences in its disjunctions. This makes it possible to use the same implementation of an FSM with different clients and clients that not always expose events the same synchronicity.

In Figure 5.6, this is used to coordinate the action events with the *state events* `ON` and `OFF`, triggered in case of a switch to the corresponding states. Since the code for transactions marks the state by triggering the associated *state event* without waiting for the consumption of the event occurrence to complete the transition, such events are defined asynchronous. Starting with the initialization of the state machine by triggering the event `OFF`, these events are systematically queued by the disjunction. They are only dequeued when the proper action event is triggered, which results in triggering, in sequence, either the event `toggleON` or the event `toggleOFF` depending on whether the first or the second Join pattern matches. The handlers bound to these events are then responsible for actually switching the state by triggering the appropriate state event.

Unlike our previous example, which used a monitor (Figure 5.5), such a switch does not block the threads delivering asynchronous events.

5.3.4 One step further

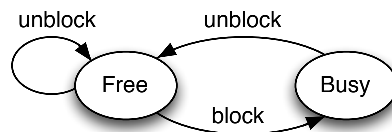


Figure 5.7 – Repeated state machine for request-rate limiter.

Figure 5.7 represents the finite state machine from Figure 3.4. The state machine shows a small variant of the previous two-state on-off state machine, modulo renaming of the states and actions: the action switching back to the initial state can now also occur in the initial state. The corresponding implementation is shown in Figure 5.8.

Before, we used this state machine in order to limit the request rate of a server. The idea is, that on the one hand, requests are blocked by default, switching the state of the machine to busy, while, on the other hand, unblocking events are generated at a constant rate, allowing one pending request to proceed and switching back the machine to free. Of course, unblocking events can also be generated while the machine is free. We cannot use the previous state machine, which does not have the transition looping on the initial state. Because this state machine does not avoid the risk of piling up unblocking events

```
1 class RateLimiter(evt block[Unit], evt unblock[Unit]) {
2   private imperative async evt Free[Unit]
3   private imperative async evt Busy[Unit]
4   private evt (toBusy, freed, absorbed)
5       = (block & Free)
6       | (unblock & Busy)
7       | (unblock & Free)
8   private evt toFree = freed || absorbed
9    toFree += ((_)=>Free() )
10   toBusy += ((_)=>Busy() )
11   Free()
12 }
```

Figure 5.8 – Implementation of concurrent request-rate limiter.

in the free state. When there are n unblocking events piled up, they unblock the first n block events at once. Which bypasses the rate limitation. A detailed description of this state machine can be found in Chapter 3.

The principle of the implementation is the same as before, with now three Join patterns in the disjunction as there are three transitions in the state machine. As two transitions lead to the same state `Free`, an intermediate event `toFree` is defined as the union of the events `freed` and `absorbed`. This event `toFree` corresponds to the two transitions that lead to the state `Free`. This is another example of a declarative event. An alternative would have been to bind the same handler, triggering an event `Free` to the events `freed` and `absorbed`.

5.4 A DSL to describe an FSM

Leveraging the regularity of FSMs to directly support them with a DSL provides opportunities to further reduce boilerplate code compared to the event-based implementation. For example, intermediate events, which are required in the native JEScala implementation to model transitions can be removed if they are not needed by the rest of the application. Extending JEScala with static bindings could reduce the code for an implementation as well. However, it does not result in an explicit description of an FSM. As we will discuss soon, the DSL also opens up opportunities for performance optimization.

5.4.1 FSM definition

```
1 Busy -> Free on unblock
2 Free -> Busy on block
3 Free -> Free on unblock
4 initialState(Free)
```

Figure 5.9 – DSL representation of a state machine.

As we have seen, the Joins-based state machines in JEScala support synchronous and asynchronous *action* events. These finite state machines can support *reactions* by registering handlers with the resulting events of patterns, which each model a transition.

To introduce our DSL, we start from the example in Figure 5.9, which shows the machine from Figure 5.7 as a number of transitions and an initial state. A transition is formally a tuple of three elements: a source state s, an action a, and a target state t. The description uses the following concrete syntax for a transition: s `->` t `on` a, where the arrow operator evokes the graphical notation for an edge between the source and the target state.

Our DSL is built around this core syntax. We additionally allow a transition to be followed by an optional reaction `triggers` e or `apply` f, where e denotes an imperative event without parameters, to be triggered as an output of the transition, and f a function without arguments, to be applied as an output of the transition. In the latter case, applying a function f is a shortcut for triggering a private synchronous event e that is associated to a single handler f.

```
1 trait S
2 object Free extends S
3 object Busy extends S
4 class RateLimiter(val cl:Client) extends FSM_J[S] {
5   imperative async evt waiting[Unit]
6   Busy -> Free on cl.unblock apply(()=>println("freed") )
7   Free -> Busy on cl.block triggers waiting
8   Free -> Free on cl.unblock
9   initialState(Free)
10 }
```

Figure 5.10 – DSL-based version of the state machine.

Figure 5.10 shows a complete program for the same state machine, with an additional

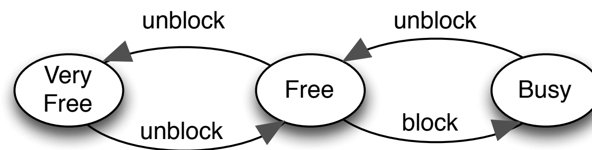


Figure 5.11 – Adding state VeryFree to the request-rate limiter FSM.

reaction for the first two transitions as an illustration. It now includes the code that makes it possible to embed the initial description into JEScala.

First, all the states are declared as Scala objects that extend the same trait, here `S`. Then, the class `RateLimiter` extends the implementation trait `FSM_J` using `S` as its type parameter.

Finally, all the events used by the state machine should be defined. These events can be part of other objects, as long as they are visible. Here, we assume that the action events `block` and `unblock` are defined in a class `Client`. The event `waiting` is an asynchronous event defined as such on Line 5. Its synchronicity prevents handlers to cause delays in the FSM. This event could drive a future logger that would record when the rate limiter starts waiting.

Specifying the initial state `free` (Line 9) concludes the class definition. This causes the description to “freeze”. Which means that the description cannot be changed any longer. Then the second layer starts generating the structures to implement the state machine that we described.

To modify state machines via inheritance, we use an alternative implementation that offers the possibility for a subclass to unfreeze the description from a superclass.

Extending FSMs via inheritance Our DSL seamlessly integrates with the inheritance mechanism of Scala classes. This makes it possible for a subclass to modify an FSM that is defined in a class. Instead of requiring the programmer to design an incomplete FSM and complete it in a subclass, we also support deleting transitions to reshape a working FSM in a subclass. When a state is split into several states, the transitions leading to the removed state must be re-targeted with the target updated to the correct new state. We insert, as an example, a state `VeryFree` into the loop of the existing FSM from Figure 5.10 by using inheritance. Figure 5.11 shows the resulting state machine. The machine cannot block in the additional state before an `unblock` event is consumed. Figure 5.12 shows how a subclass insert a state `VeryFree` into the loop of the existing FSM (Figure 5.10). This results into the state machine that we show in Figure 5.11.

```
1 object VeryFree extends S // additional state
2 class Modified(val cl:Client) extends RateLimiter(cl) {
3   unfreeze
4   remove Free -> Free on cl.unblock
5   Free -> VeryFree on cl.unblock
6   VeryFree -> Free on cl.unblock
7   initialState(Free) // creates the modified FSM
8 }
```

Figure 5.12 – Modifying the FSM by inheritance.

The additional state (Line 1) needs to be defined similarly to the existing ones (`Free`, `Busy`).

First the subclass needs to `unfreeze` (Line 3) the definition of the superclass. It can remove existing transitions by specifying the source state, destination state and the *action* event of an existing state, in other words repeat the existing transition after `remove`. Adding new transitions (Line 5) is similar to defining transitions in the description of the superclass. This subclass inserts a state `VeryFree` into the loop of the existing FSM.

5.4.2 DSL design

We adopt a two-layer DSL: An abstract trait `FSM` provides a generic infrastructure for collecting the transitions of the state machine from the description. Specific implementations trait extend the `FSM` trait with specific code that is responsible for actually building the state machine. For instance, the implementation trait `FSM_J` uses Join patterns to build a state machine based on some of the ideas presented in Section 5.3. These two layers make it possible to provide different concrete implementation of the FSM DSL.

An implementation trait inherits from the trait `FSM`. The trait is responsible for generating a state machine from its transitions. For example `FSM_J` generates an FSM semantically equivalent to Figure 5.8. Each description that uses the DSL end with a call to the `initialEvent` method from the implementation trait. The call to the method `initialEvent` triggers the trait `FSM_J` to create data structures via the underlying library from JEScala instead of using the syntax offered to the programmer. The structures are similar to what the programmer creates manually. The defined reactions are implemented as additional handlers to the resulting events of Join patterns.

Optimization Using Joins has the advantage that it does not make assumptions on the synchronicity of the *action* events. However, the state events are always asynchronous and a state machine has always one active state, that is modeled by one triggered *state* event. Events support modularity via implicit invocation and dynamic registration. However, the *state* events are only observed by a single disjunction that is part of the object defining the state events. The native implementation of disjunctions in JEScala needs to deal with varying synchronicity and the events have to support implicit invocation. In alternative implementation traits for the DSL we use another representation for states.

We combine parts of the JEScala implementation of the disjunction with conventional implementations of state machines. With respect to the *action* events, the alternative implementation is similar to the Joins. With respect to the state events we use an alternative implementation. This implementation directly implements the transition function of the state machine, with two different variants that we explain in Section 7.4. Our first implementation uses a hash table for the transition function. Because we can number both the states and the action events, we implemented an alternative based on a two-dimensional array.

5.5 Advanced State Machines

We developed in the previous chapter a DSL to describe FSMs that react to action events without arguments. Restricted action events have the advantage of a simple syntax for the DSL. Such FSMs are useful to express coordination, as in the case study (Chapter 3) that initially motivated the study for the DSL. If the events that drive a state machines have unnecessary arguments, it is possible to remove the arguments by using `dropParam` in a declarative event expression. However, some applications rely on an FSM receives arguments via the action events. First, we study in this chapter an example inspired by functional reactive programming (FRP) with Flapjax [83]. The event *streams* from FRP are comparable with the events in JEScala. While the event streams are not object members, we use declarative JEScala events to represent them. Second, we show that some state machines do not have a fixed amount of states. Such parameterized state machines adapt after they are designed to the environment they are used, for example to coordinate a set of threads that is only known at runtime.

```
case class Point(x:Int, y:Int) {}

object Mouse {
  imperative async evt mousePos[Point]
  imperative async evt mouseButton[Boolean]
  ..
}
```

Figure 5.13 – Mouse object events.

5.5.1 Action events with arguments

We show how a state machine that reacts to action events with arguments makes it possible to implement an FRP example in JEScala.

A mouse as a source for events Graphical User Interface (GUI), the mouse is a source of two kinds of events: position events and mouse button events. For our example, we consider a mouse with a single button that sends via an event two positions of the button (*up* and *down*). The position is represented by a Boolean argument of the event.

We represent this mouse as a singleton object. This representation reduces the size of the code by avoiding initialization code to construct objects from a class. Lines 3–7 in Figure 5.13 illustrate the JEScala implementation of a `Mouse` object. The two asynchronous events `mousePos` (Line 4) and `mouseButton` (Line 5) are the inputs for the `mouse` object. The `mousePos` event emits the current position when the mouse moves. The `mouseButton` event emits the position of the button after pressing or releasing the single mouse button.

Dragging objects Dragging means pressing the button when the mouse pointer is above a graphical object on the screen to move the object, until the button is released again. Therefore, we only need the position of a dragged object when the button is pressed. For a simple implementation we ignore the discovery of the underlying object when the mouse is pressed.

The original example creates a new event stream each time the mouse button is pressed. It is possible to mimic this in JEScala by creating an object that contains a drag event. After releasing the mouse button the object can be discarded. However, we prefer to expose the dragging information about an object via an event that is a member of the

```
1 case class Point(x:Int, y:Int) {}
2
3 object Mouse {
4    imperative async evt mouseButton[Boolean]
5    var buttonState=false
6    def storeState(arg:Boolean):Unit = {
7       buttonState=arg  }
8    mouseButton += storeState _
9    imperative async evt mousePos[Point]
10 }
11
12 object Draggable {
13    evt dragged=Mouse.mousePos && ((_) => Mouse.buttonState )
14 }
```

Figure 5.14 – Naive filtering of `mousePos` events.

object. This makes it possible for any observer to register with the event from any object that can be dragged.

Naive implementation In Figure 5.14 we show a naive implementation, which does not protect the shared `buttonState` variable. The code starts with defining a case class `Point` (Line 1), which is a Scala construct to create an immutable container for two coordinates. The constructor arguments `x` and `y` are implicitly defined as public read-only values. The `Mouse` object (Line 3) follows with a declarative event expression that emits position events when moving with a pressed mouse button. Because the position of the mouse button is only available when receiving a `mouseButton` event, the handler of this event stores the position in a variable `buttonState` (Line 7) of the `Mouse` object. However, this variable is accessed via two threads that deliver the two asynchronous events, which requires synchronizing the accesses (Lines 7, 13) to the `Mouse.buttonState` shared variable (Line 5).

Thread-safe implementation The thread-safe alternative in Figure 5.15 represents the position of the mouse button by an FSM with the two states `pressed` and `unpressed`. The action events `buttonDown` and `buttonUp` change the state. We show in the code of the `Mouse` object (Figure 5.16) that JEScala can extract two events from a single event with a Boolean argument for the mouse position, by using declarative expressions. This state machine can handle the action event `mousePos` in each state, but these

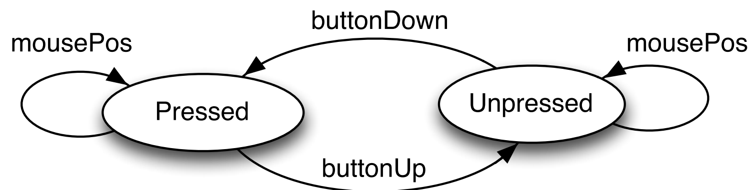


Figure 5.15 – Mouse button as a state machine.

transitions loop: the final state is the same as the initial state. In other words these action events do not change to a different state. However, we need to treat `mousePos` as an action event, because its argument of the action events carry the position of the mouse. Each of the two states handles the action event in a different way. When we ignore the loops in each state, the FSM is comparable to the switch FSM from Figure 5.1, except for renaming the states`ON` and `OFF` to respectively `pressed` and `unpressed`. Different from the switch FSM is that to prevent codemousePos event occurrences from piling up in any state, the FSM consumes `mousePos` events in both of its states. Similar to the consumption of `unblock` events in Figure 4.4b, we model this as a Join pattern that combines a state event and an action event, which has in this case a `Point` argument. Both resulting events contain the arguments from both joined events. However, the resulting events need to trigger the resulting state event without an argument. In other words the resulting state event is the same state that causes a Join pattern to match. The resulting events from both join patterns carry the parameter from the `mousePos` event combined with the empty parameter of the state event. A declarative event expression recovers the parameter received via a `mousePos` event.

Figure 5.16 first repeats the definition of the `Point` case class. It shows then an adapted `Mouse` object with the declarative `buttonUp` and `buttonDown` events, followed by an implementation of the `Draggable` object that contains the state machine from Figure 5.15 described above. The state machine from Figure 5.15 extends the RateLimiter shown in Figure 5.7 with a looping transition to the state `Busy`, which had only a transition to the `Free` state and renames the states and actions. We show below in the implementation that the action events of both looping transitions have an argument that indicates the mouse position.

The state machine controls the emission of mouse positions via the declarative event `dragged` on Line 23 by using the mouse button. This code is similar to state machine shown in Figure 5.8, except for the `mousePos` action event that causes a looping transition in both states. The event has a `Point` argument. An alternative implementation could encode the state (pressed, unpressed) as a boolean argument of a single state event. Such implementation would use a single handler with an `if` statement to trigger

```
1  case class Point(x:Int,y:Int) {}
2
3  object Mouse {
4    imperative async evt mouseButton[Boolean]
5    evt buttonDown=(mouseButtom && ((down:Boolean)=>(down))).dropParam
6    evt buttonUp=(mouseButtom && ((down:Boolean)=>!(down))).dropParam
7    imperative async evt mousePos[Point]
8  }
9
10 object Draggable {
11   imperative async evt pressed[Unit]
12   imperative async evt unpressed[Unit]
13
14   evt (down, up, useful, absorbed)=
15         unpressed & Mouse.buttonDown |
16         pressed   & Mouse.buttonUp   |
17         pressed   & Mouse.mousePos   |
18         unpressed & Mouse.mousePos
19   evt toDown= down || useful
20   evt toUp=   up   || absorbed
21   toDown += ((arg:Any) => pressed() )
22   toUp   += ((arg:Any) => unpressed() )
23   evt dragged= useful map ((a:(Unit, Point)) => a._2 )
24   unpressed()
25 }
```

Figure 5.16 – Filtering mouse position events to drag-only position events.

the single state event with the right value. The disjunction of such implementation has only two patterns instead of four. However, the implementation would have no explicit state events. We show in Section 5.5.2 an example that encodes several states in a single state event with an argument.

The object `Mouse` models the input from the mouse via two imperative events: `mousePos` and `mouseButton`. The events `buttonDown` (Line 5) and `buttonUp` (Line 6) refine the single `mouseButton` (Line 4) which indicates the position of the button by a Boolean argument.

The object `Draggable` observes events from the `Mouse` object. As we mentioned, the code does not include any logic to know whether a mouse click occurs above any `Draggable` object. However, a complete implementation has to include such logic to make it possible to move only the graphical object under the mouse pointer instead of moving all graphical object together.

The `Draggable` object defines the two state events `pressed` (Line 11) and `unpressed` (Line 12). The disjunction on Lines 14–18 implements each transition of the state machine in a binary pattern, which joins a state event with an action event. There are two kinds of resulting events in this FSM. The resulting `down` and `up` events change the state, unlike the events `useful` and `absorbed` that loop back to their state. The events of the patterns that implement the looping transitions contain a position argument from the `Mouse.mousepos` action event. The event `dragged` (Line 23) extracts the position that was received via the `Mouse.mousepos`. The `absorbed` event is only used to return to the `unpressed` state without using its argument. The events `toDown` (Line 19) and `toUp` (Line 20) indicate the target state after a transition, which is similar to the `toFree` and `toBusy` in the `RateLimiter` class, that was shown before in Figure 5.15. To explicitly trigger the event for the new state register handlers with the events `toFree` and `toBusy`. on the Lines 21–22. The initial goal of this FSM is to define the event `dragged` (Line 23) that emits the values from the `mousePos` events when the button is pressed. This requires extracting the argument of the second event from the resulting Join event. This event is the result of combining the mouse pressed state with a mouse move action. To conclude, the constructor initializes the finite state machine by triggering the `unpressed` state event at Line 24.

5.5.2 State events with an argument

An argument for one or more state events can represent additional states. As an example, we show a reader/writer lock. This lock protects shared variables that are accessed by multiple concurrent clients. More than one client can safely concurrently read these variables. However, to change one of these variables, a client has to gain exclusive

access before changing that variable, to ensure no other client reads a variable in an inconsistent state.

Figure 5.17 shows an FSM to model this lock. We can implement this as an FSM using the DSL we defined before as shown in Figure 5.18. Before a writer accesses the shared variable it needs to wait until the synchronous `wLock` event returns. The writer triggers the `wUnlock` event after it completed its task. Readers need to comply with a similar protocol, except that there are multiple `Read` states (Lines 4–5) that each represent a different number of active readers (e.g. 1 or 2). When the states are represented with asynchronous imperative events that have no argument, an additional state event is required to represent each number of concurrent readers. Extending the implementation of the lock with an additional concurrent reader requires a new state and two transitions.

The implementation in Figure 5.19 has the same semantics as the one described in Figure 5.18. However, this implementation uses the single asynchronous `readerCountState` event (Line 8) with an `Int` parameter instead of using a dedicated state event for each number of concurrent readers.

5.6 Conclusion

First, we gave an overview of state machines in general. Then we showed our DSL for FSMs without arguments. The language ECaesarJ [88] does also provide the programmer with language support for event-driven finite state machines without concurrency. However, it groups actions for each state, where our DSL describes transitions more explicit.

After this, we showed two examples that our DSL for FSMs cannot describe. The first uses an action event with a parameter, the second has an *almost* infinite number of states. We use the word *almost* since Scala represents signed integers with 32 bits, which limits the number of different `read` states. However, the examples represent the number of

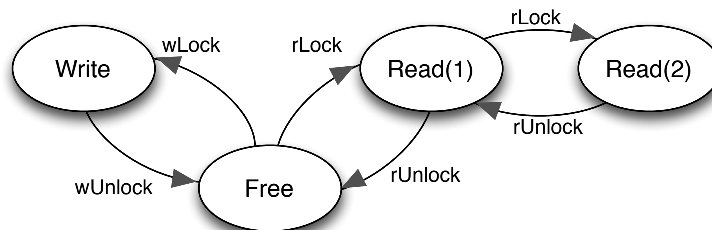


Figure 5.17 – FSM of a reader/writer lock with at most two concurrent readers.

```
1 trait State
2 object Write extends State
3 object Free  extends State
4 object Read1 extends State
5 object Read2 extends State
6 class RWLock1 {
7   imperative sync evt wLock[Unit]
8   imperative async evt wUnlock[Unit]
9   imperative sync evt rLock[Unit]
10  imperative async evt rUnlock[Unit]
11  Free   -> Write on wLock
12  Write  -> Free  on wUnlock
13  Free   -> Read1 on rLock
14  Read1  -> Free  on rUnlock
15  Read1  -> Read2 on rLock
16  Read2  -> Read1 on rUnlock
17  initialState(Free)
18 }
```

Figure 5.18 – Implementation of reader/writer lock with at most two concurrent readers via DSL.

```
class RWLock2 {
  imperative sync  evt wLock[Unit]
  imperative async evt wUnlock[Unit]
  imperative sync  evt rLock[Unit]
  imperative async evt rUnlock[Unit]
  imperative async evt writeState[Unit]
  imperative async evt free[Unit]
  imperative async evt readerCountState[Int]
  evt (toWrite, toFree, toRead, toDecrRead, toIncrRead)=
      free              & wLock    |
      write             & wUnlock  |
      free              & rLock    |
      readerCountState  & rUnlock  |
      readerCountState  & rLock
  toWrite    += ((_)=> write() )
  toFree     += ((_)=> free()  )
  toRead     += ((_)=> readerCountState(1) )
  toIncrRead += ((s,_) => readerCountState(s+1) )
  toDecrRead += ((s,_) => if (s>0) readerCountState(s-1) else free() )
  free() // initial state
}
```

Figure 5.19 – Implementation of a reader/writer lock without an upper bound on concurrent readers.

concurrent reading threads by a state and we do not expect the total number of threads to exceed 1000 active threads on a JVM, which is only a fraction of the upper limit ($2^{31} - 1 \approx 2 \cdot 10^9$) that our implementation supports.

Further study could result in a new DSL that is able to describe also the state machines that our current DSL cannot describe.

Event Monitor

Contents

6.1 Example **132**
6.2 Synchronizer **134**
6.2.1 Redesigning the synchronizer 134
6.2.2 The synchronizer as an actor 135
6.2.3 Bridging the gap 136
6.3 Limitations and future extensions **136**

Thread-based concurrency in Java often uses monitors. While monitors are from the beginning a part of Java, the development of concurrent software did not become easier. One of the reasons that monitors are hard to use is understanding and using the `wait` and the `notifyAll` operations in a correct way. These operations are needed to control the interaction of threads that access shared data.

The handlers in the JEScala event system are not protected against race conditions, because they can access shared data. Restricting the handlers to pure functions results in handlers that cannot cause race conditions, but the execution of pure functions cannot change the state of the program. Moreover, the result of their calculation is lost, which means the compiler could omit every call to such functions as an optimization.

We experiment in this chapter by organizing handlers in groups to prevent concurrent execution of the handlers that belong to the same group. By using asynchronous events, delaying the execution does not block the sender of such events, since blocking senders of events can lead to deadlocks, when other code directly or indirectly waits for the sent events. By restricting the concurrency among handlers, we can reuse code that is designed for sequential execution in a concurrent program.

Of course, mutual exclusion among handlers could be enforced by using the synchronized blocks that Scala inherits from Java. However, handlers that are part of a mutually exclusive group need to synchronize with a single object. Because the registration of handlers can be spread over different parts of the code that are designed independently, and because JEScala supports dynamic registration one handler does not always belong to the same group. When using synchronized blocks, this flexibility can lead to accidentally using different objects to synchronize handlers. Another disadvantage of using synchronized blocks, is that when handlers are waiting to be executed, their executing threads remain occupied. When a program only needs to execute a handler eventually, but the program can continue without waiting for the execution, this handler can be executed asynchronously. Such handlers do not need to block a thread for their eventual execution.

```
1  class EscalaUpDown {
2    var state:Int= 0
3    imperative evt sync up[Int]
4    imperative evt sync down[Int]
5    def inc(arg:Int):Unit= {
6      val old= state
7      // unsafe to interleave
8      state= old+arg
9    }
10   def dec(arg:Int):Unit= {
11     val old= state
12     // unsafe to interleave
13     state= old-arg
14   }
15   up += inc _
16   down += dec _
17 }
```

Figure 6.1 – EScala example.

Figure 6.1 combines in a single class the handlers `inc` (Line 5) and `dec` (Line 10) that

```
class JEScalaUpDown {
  var state:Int=0
  imperative evt async up[Int]
  imperative evt async down[Int]
  imperative evt async lockFree[Unit]
  def inc(arg:Int):Unit= {
    val old= state
    state= old+arg
    lockFree()
  }
  def dec(arg:Int):Unit= {
    val old= state
    state= old-arg
    lockFree()
  }
  evt (safeUp,safeDown)=
        (lockFree,up)
      | (lockFree,down)
  safeUp += inc _
  safeDown += dec _
  lockFree()
}
```

Figure 6.2 – Adding concurrency to the EScala example.

both modify the shared variable `state` (Line 2). The expression in the handlers is split in two steps to show where a possible race condition could appear when using concurrency without proper synchronization. Because the example uses synchronous events, it is a valid EScala program when we remove the `sync` modifier of both imperative events.

We can reuse existing EScala handlers in a JEScala program with asynchronous events like shown in Figure 6.2 registering the same handlers with asynchronous events. This implies a need for additional synchronization code. Of course, we can use a disjunction on Line 16 and the asynchronous event `lockFree` (Line 5) to prevent race conditions between the two handlers. This code explicitly triggers the `lockFree` event at the end of each handler, which can be replaced by an implicit event. Our example combines all parts into a single class, but events, handlers and their registration can be in different parts of a program. Adding a third handler that cannot run at the same time as the `inc` or `dec` requires adding a Join pattern to the disjunction.

When extending the code in Figure 6.2, the complexity grows, especially when handlers

are part of different objects. The goal of using disjunctions and an asynchronous event is to ensure that a group of handlers that access a shared resource do not interfere with each other. Therefore we develop a component that acts as an interface between the group of mutually exclusive handlers and the events to which they are associated. We call our implementation a *synchronizer*, which provides an interface between the events and each handler of a group of handlers. This interface component organizes all event occurrences sequentially. The double bang operator (!!) can be used to make sure that all event occurrences become asynchronous, before they are put in a queue to be executed. After we studied this component it became clear that our implementation creates a monitor for events, hence we consider this an implementation of an *event monitor*. This name indicates that they are related to the Sequential Object Monitors [21], which handle method calls to a group of methods sequentially.

6.1 Example

In a traditional bank, a client fills out a form at the counter and signs it. Clients need to repeat this for each operation. The transactions that result from these forms are later executed, usually during the night.

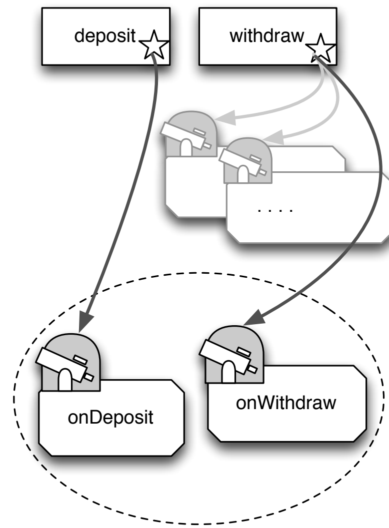
```
1  object Counter {
2    imperative evt async deposit[Double]
3    imperative evt async withdraw[Double]
4  }
5  class Account {
6    var amount:Double=0
7    def onDeposit(a:Double)= {amount=amount+a }
8    def onWithdraw(a:Double)= {amount=amount-a }
9    Counter.deposit += onDeposit _
10   Counter.withdraw += onWithdraw _
11 }
```

Figure 6.3 – Handlers are not more safe than other methods.

Figure 6.3 models this example using JEScala. Because transactions are executed at night, we declare the imperative events as asynchronous so that the execution of the handlers does not block the code triggering the events. The code ignores that the handlers for these asynchronous events are not supposed to execute the transactions before the night. For simplicity, both events are part of a single counter. In practice, they would be extracted from all the counters and associated with a given customer.

```
1  object Counter {
2    imperative async evt deposit[Double]
3    imperative async evt withdraw[Double]
4  }
5  object Synchronizer {
6     def register[T](ev:Event[T],
7                     handler:(T=>Unit) ) =
8        { ... }
9  }
10 class Account {
11    var amount:Double= 0
12    def onDeposit(a:Double)= {
13       amount= amount+a
14    }
15    def onWithdraw(a:Double)= {
16       amount= amount-a
17    }
18    Synchronizer.register(Counter.deposit,
19                          onDeposit)
20    Synchronizer.register(Counter.withdraw,
21                          onWithdraw)
22 }
```

(a)



(b)

Figure 6.4 – Handlers mutually exclusively executed.

The first step to avoid data races on accounts is to group the handlers that should not execute concurrently because they access shared data. We assign a *synchronizer* component to such groups, which ensures sequential execution of handlers in the group. Figure 6.4a shows the interface of a *synchronizer* component that registers handlers with events and prevents the handlers to interleave. The synchronizer contains a single method to register handlers that belong to a single group with events. For this example, we show the `Synchronizer` as a Scala object, which defines a single group. Figure 6.4b presents the handlers as telescopes that observe stars as symbols for events. An interleaving of the handlers `onDeposit` and `onWithdraw` may result in a race condition. We want to group these handlers to ensure that they are executed after an occurrence of the event they observe, without any interleaving. The *synchronizer* groups handlers: while a handler can observe different events: the synchronizer prevents race conditions among the handlers that are registered via the same synchronizer. It is still possible to register

handlers with the += operator to execute like a normal JEScala handler.

6.2 Synchronizer

We introduced the synchronizer as a way to group handlers that access one or more shared variables. The goal of defining this group is to make sure that these do not concurrently execute. Handlers registered to the same group can safely access shared variables.

By restricting the concurrency among the handlers of a group it is possible to use handlers that were designed for sequential EScala programs in a concurrent way. Concurrency among the handlers grouped by a synchronizer is restricted. However, a program can define multiple groups each with their own synchronizer.

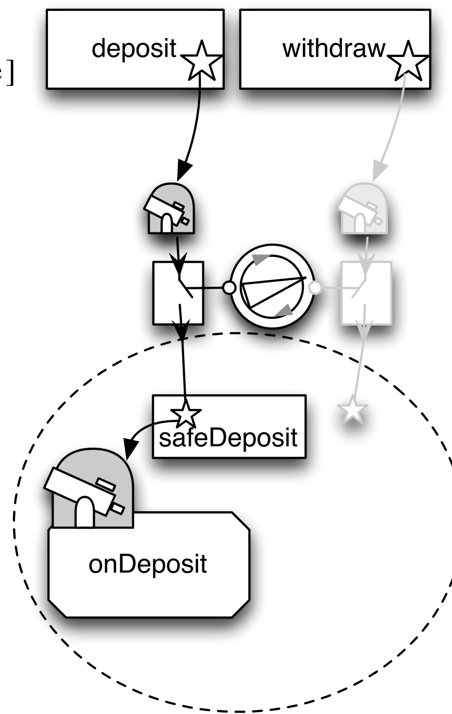
Our first experiment uses a specific registration system for handlers that are grouped by a synchronizer. In the following, we come back to a standard registration mechanism by changing the interface of the synchronizer, which makes it possible to associate new "synchronized events" to the events whose handlers need to be synchronized. Once this is done, registering the handlers with the synchronized events rather than with the initial events guarantees that the events are synchronized.

6.2.1 Redesigning the synchronizer

We redesign the synchronizer to a component that observes events and provides a synchronous event for every observed event. Handlers can register with the created synchronous events like with normal events. However, the synchronizer that created the synchronous events triggers the synchronous events in a sequential way. As a result the associated handlers are executed sequentially, as the programmer does not trigger any asynchronous events. Figure 6.5a shows the API for the redesigned synchronizer. The reduced example shows only a single event handler, while there are several handlers like before. The synchronous event `safeDeposit` is triggered after the event `deposit` from the `Counter` object is triggered. The central part in Figure 6.5b behaves like a revolving door that lets a single event occurrence flow from the original event (`deposit`) to the synchronous event generated by the synchronizer (`safeDeposit`), the second event inside the collection of handlers represents a similar `safeWithdraw` event, without showing any details of it.

```
1  object Counter {
2    imperative async evt deposit[Double]
3    imperative async evt withdraw[Double]
4  }
5  object Synchronizer {
6     def getProxyEvent[T](
7          ev:Event[T]
8          ):ImperativeSyncEvent[T] =
9             { ... }
10 }
11 class Account {
12    evt safeDeposit=
13       Synchronizer.getProxyEvent(
14             Counter.deposit
15             )
16    safeDeposit += onDeposit _
17 }
```

(a)



(b)

Figure 6.5 – Synchronizer: A practical implementation.

6.2.2 The synchronizer as an actor

The enhanced design from Section 6.2.1 shows that the event occurrences reach the group of handlers sequentially when handlers only register to events provided by the synchronizer. The sequential handling of event occurrences is similar to sequential handling of messages in an actor. However, there are differences. For instance, the synchronizer exposes events with implicit invocation to a group of handlers, while sending a message requires that the receiving actor is known to the sender.

The way actors handle messages is part of the design of an actor. Instead, the synchronizer presents events, to which handlers can register dynamically. While actors are designed for a specific task, the synchronizer is a reusable component. By grouping the handlers that access a shared resource, it is possible to reuse existing parts of sequential code in a concurrent program.

The design of an actor defines to which messages it reacts. However, the synchronizer has to be flexible to deal with events that are not known before run time, since they can

be dynamically chosen at runtime. The flexibility that originates from JEScala's event system supports dynamic registration for handlers.

The messages can also be sent to the synchronizer asynchronously but the use the `!!` operator turns them into a synchronous events. Actors use a single thread to handle messages. A single thread is enough for a synchronizer to execute the handlers, since concurrent execution of the handlers is undesirable.

When programming with actors, the user can break the model by explicitly creating concurrency inside the actor. Similarly, creating concurrency among the handlers that are grouped by a synchronizer is possible by using asynchronous events, the programmer is responsible for not creating concurrency.

6.2.3 Bridging the gap

Actors handle the messages they receive sequentially. While actors use messages, the synchronizer uses JEScala events. By using the underlying Scala language we can implement the synchronizer with actors. We explain in Section 7.5 our implementation, which serializes JEScala event occurrences via the mailbox of a Scala actor and afterwards emit these occurrences via a synchronous event with arguments of the same type as the serialized event occurrence.

6.3 Limitations and future extensions

The synchronizer has no equivalent to the `wait` and `notifyAll` operations, handlers execute without interleaving, unless they throw an exception.

We describe the key points of our implementation of the synchronizer as an actor in Section 7.5. We implemented most of the functionality in an abstract trait. The synchronizer extends from this trait and implements a concrete actor. In addition it is possible to build other extensions of the abstract trait, which can be used as a Scala actor that still synchronizes events but can react to messages. However, we have not explored such extensions that combine JEScala events with Scala actors.

When we combine an actor with a synchronizer, it is still possible to create a controlled form of concurrency in case the application needs this. For example, to control an active task that is implemented as a handler. If concurrency is needed, the task of a handler can be split in multiple atomic steps, implemented as *sub-handlers*. The sub-handlers can use actor messages to invoke their successor. This is a way to control the interleaving between the handlers that were split. This makes controlled concurrency

possible similar to cooperative multitasking in some operating systems. In other words, programs give up their execution at well-defined points of their execution.

Implementation

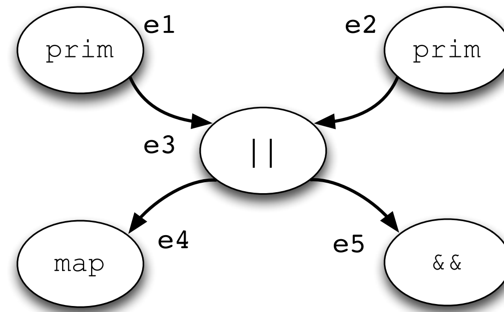
Contents

7.1 Event graphs . . . 140
7.1.1 Triggering an event . . . 141
7.1.2 Dynamic deployment . . . 142
7.2 Adding asynchronous event handling . . . 143
7.3 Handling disjunctions . . . 143
7.3.1 Optimizations . . . 144
7.4 FSM DSL . . . 145
7.5 Event Monitors . . . 147

The implementation of JEScala required us to modify the EScala event system to support Joins and asynchronous events. Currently, JEScala is implemented as a library. Future work could integrate this library into Scala by modifying the compiler as it was done for EScala. Before describing how the events with different synchronicity are implemented, we summarize the internals of the EScala event system. Further details about EScala are described in [44]. We explain the implementation of a disjunction and how JEScala can optimize a disjunction in specific cases. Then we explain the underlying implementation of the DSL that we designed to describe finite state machines (FSM). To

```
1 imperative evt e1[Int]
2 imperative evt e2[Int]
3
4 evt e3= e1 || e2
5
6 evt e4= e3 map ((x)=> (x*2) )
7 evt e5= e3 && ((x)=> (x>2) )
```

(a)



(b)

Figure 7.1 – An event graph is not always a tree.

conclude, we show we can implement an event monitor, which is a monitor-like system to group handlers in a way that they cannot execute concurrently.

7.1 Event graphs

Declarative event expressions are used as a description of a graph with the shape of its abstract syntax tree. However, the combination of event expressions can result in a graph that is not a tree. The event graph is a central part of the event system of EScala and was slightly modified for JEScala. Some optimizations in EScala and JEScala rely on the event graph.

Figure 7.1a shows an EScala program that only declares events. These declarations result into the event graph shown in Figure 7.1b. Let us explain the code together with the structure of the matching graph in order to give a feel for the implementation. Note that the graph is constructed with objects from different types, shown by circles that contain a label to show their type. We use the same notation as the EScala code, instead of using the real class names from the implementation. The imperative events are represented as primitive events. These are the same type of event nodes that are used for implicit events.

Lines 1 – 2 create the two primitive nodes on top of Figure 7.1b. Because the events `e1` and `e2` are primitive events, their nodes have no incoming edges. Line 4 defines the declarative event `e3` as a union of the events `e1` and `e2`. The central node in the graph represents this declarative event. The nodes `e1`, `e2` and `e3` represent the abstract syntax tree for the expression that defines `e3`

The event e3 is the input for the two declarative events e4 and e5. Because of this, the event graph in Figure 7.1b is not a tree. Line 6 defines e4 as a transformation of e3. The function on the right of the map operator doubles the parameter of the event occurrences of e3 to create the event occurrences of e4. The node with the map operator represents event e4. Line 7 defines event e5, which uses the same event e3 as input. The operator && filters the event occurrences of e3. When the parameter of an event occurrence of e3 satisfies the predicate x=>(x>2) at the right side of the && operator, an event occurrence of e5 is triggered.

7.1.1 Triggering an event

Event nodes can have two kinds of observers: the handlers registered with the node and its sinks, that is, the event nodes that observe the node. To store both kinds of observers each event node contains two dedicated collections.

Triggering primitive events requires an argument v of type T, holding the data carried by the event occurrence. If the argument has type Unit, the event can be seen as not carrying any data. Triggering an event creates an empty collection of *reactions*, functions of type () => Unit and accumulates in this collection all the reactions encountered starting with the event node associated to the event. Each encountered event node creates for each of its handlers h, a new reaction () => h(v) and requests its sinks to do the same. This results in a depth-first traversal of the active part of the event graph. After traversing the graph, this collection contains all the reactions associated to the primitive event occurrence. The two-phase event handling executes the collected reactions only after the collection phase completes, which prevents modifying the collection of reactions during their execution.

Event nodes defined by the map and filter (&&) operator perform an additional step before continuing to traverse the graph in order to extend the collection of *reactions*. The right operand of both operators is a function (f(v:T) => S). For the filter operator, this operand is a predicate function, in other words, its result type S is a Boolean. When the traversal of an event graph reaches a map event node, this node first applies the function f(v) to the argument v of the event occurrence and continues with the result of this function as the argument. A filter node applies its predicate to the argument of the received event occurrence. If the predicate is not satisfied, the traversal of the node immediately returns. When the programmer uses impure functions for any of these operators, the event propagation can influence itself via side effects. Another reason to prefer pure functions is that the order in which these functions are executed is in general hard to predict, which can lead to unexpected results.

EScala refers to the phase of collecting reactions as *matching*, which can be confusing

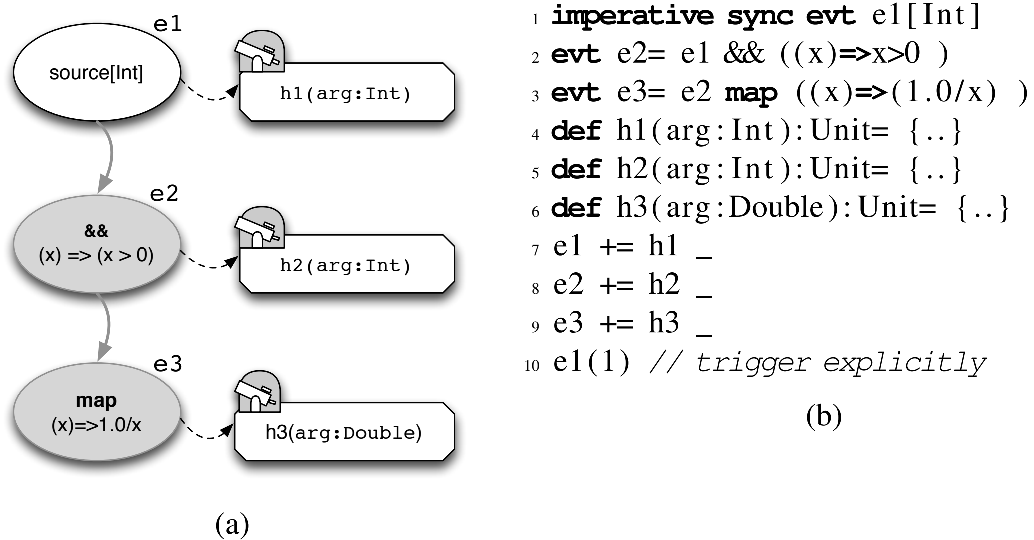


Figure 7.2 – Deployment of event nodes.

when talking about matching Join patterns.

7.1.2 Dynamic deployment

Disabling event nodes prevents the unnecessary propagation of event occurrences. The dynamic enabling and disabling of nodes in the graph can change each time the programmer registers or unregisters a handler. Figure 7.2a shows the graph for the example code shown in Figure 7.2b. The relations between an event node and a handler are shown as dashed arrows. Arrows indicate the direction in which event occurrences flow. The event node for the imperative event `e1` is a source of event occurrences resulting from triggering `e1` We ignore for the following explanation that this event node becomes inactive when it has no observers. The handlers `h1`, `h2` and `h3` (marked with telescopes) can register and unregister dynamically with the respective events `e1`, `e2` and `e3`. Event occurrences enter the graph only via event node `e1`. The event nodes `e2` and `e3` only need to receive event occurrences when they are part of a path from `e1` to a registered handler (`h2` or `h3`). If event node `e3` is inactive, registering `h3` with the event `e3` enables this event node `e3`. After enabling the event node `e3`, this iterates towards the nodes for the imperative events, until an active event node is encountered. Unregistering the last handler from an event node disables the node and follows the path towards `e1` disabling every node that has no active observing nodes (sinks), nor registered handlers, until a disabled node is reached.

7.2 Adding asynchronous event handling

In JEScala, collecting handlers (and turning them into reactions) applies to both synchronous and asynchronous primitive events. Differently from EScala, when the triggering event is asynchronous, a new thread is used to execute the collected set of handlers. As a result, the code that triggered an event and the handlers that are associated with the triggered event can execute concurrently. This also means that handler collection for several event occurrences can execute concurrently on the same graph, which requires us to make the process thread-safe. In particular, each new thread has access to its own buffer to collect handlers. Dynamic handler registration also requires that the handlers associated to an event are protected against concurrent accesses.

Unlike in EScala, the handler collection in JEScala, which we explained as a graph traversal in Section 7.1.1, propagates a flag to indicate whether the triggered primitive event is synchronous or asynchronous. This flag does not affect the EScala-like handlers, but is required to decide on storing the delivering thread in a disjunction (Section 7.3) and it helps optimize disjunctions (Section 7.3.1).

7.3 Handling disjunctions

Disjunctions are implemented as sets of queues $q_1...q_n$, a queue for each event $e_i \in e_1...e_n$ that appears in a pattern of the disjunction. When an event e_i is triggered, the event with the associated arguments is stored in the queue q_i (1) and we check if a pattern can be completed with the new event (2). If none of the patterns can be completed, e_i remains in q_i. If a single pattern matches, the associated events are removed from the queue and the resulting event is fired. If multiple patterns match, one is chosen non-deterministically and the associated events are removed.

Synchronous events that appear in a disjunction require a few additional steps. If the event is fired and none of the patterns applies, we do not only store the arguments, but we also block the thread and store it in the queue. These blocked threads return only after the event occurrences they delivered took part in matching a Join pattern and all handlers from its resulting event returned.

When all events in a matching pattern are asynchronous, another thread executes the handlers of the resulting event, similar to an asynchronous event. When one or more synchronous event occurrences are part of the match, we avoid this other thread by using the waiting thread of the syntactically first event of the pattern. For the programmer this choice is useful when accessing multiple times (before and after the pattern is selected) a library that needs to be accessed always by the same thread (e.g., the AWT library in

Java). To achieve a form of fairness, we randomly select among the patterns that can match when an event arrives. As a result our implementation depends on the fairness of the Scala random function.

The queues are new nodes in the event graph. By default, when a queue node is encountered during the collection step, queuing is not handled at once but postponed to handler execution by creating a new handler responsible for this queuing task. This is necessary so that synchronous events do not block the collection of the handlers.

7.3.1 Optimizations

Thread pool When triggering an asynchronous event, a separate thread executes all collected handlers. We expect that most handlers are short tasks that might cause new events. These separate threads are often short-lived threads, since they only execute the handlers of a single event. Thread creation is an expensive operation, therefore the use of a thread pool can reuse existing threads to increase performance.

Synchronous handler collection The implementation of asynchronous primitive events only needs a separate thread if there are some handlers for the event occurrence. Similar to the implementation of EScala, triggering an asynchronous primitive event causes a graph traversal. During the traversal, the reachable handlers are collected. Unless this collection is empty, another thread executes the collected handlers. In particular, this avoids the need for a thread in case a filter event node prevents an event occurrence from reaching any handler.

Disjunction Only When an asynchronous primitive event is only observed by disjunctions, it is possible to avoid adding handlers that only enqueue the argument of the event into a queue associated with the observed event. Instead of providing a handler during the phase of handler collection, we immediately enqueue the argument. Because enqueueing arguments takes only a short time and does not block, it is safe to do this during the collection of the handlers. When the only direct and indirect observers of a primitive event are disjunctions, the handler collection completes without any handler, which means the execution of the handlers is not needed, similarly to the previous optimization.

Replacing an event queue by a counter The ScalaJoins library provides the programmer with a faster implementation for the event queue of asynchronous events without arguments. A disjunction in a ScalaJoins object can only observe the events that

belong to the same object. As a result, the fixed one-to-one relation (a channel) between the sender and the receiver in ScalaJoins does not impose a specific location for the queuing of event occurrences. By implementing the event queue in the sender, ScalaJoins makes it possible to use an alternative implementation for an event queue, by choosing another type for the sending event.

In JEScala, there are at least four reasons to use a different approach. First, events can have multiple observers, some of them can be disjunctions that consume event occurrences at a different rate. This requires a dedicated event queue for each disjunction. Second, after permanently breaking the connection between an event and a disjunction (e.g., by a filter node in the event graph), it is possible that the disjunction has to consume event occurrences that were emitted before the connection broke. This requires the disjunction to access the event queue. Third, when a composed event is a union of two or more events, a single queue makes it possible to receive the event occurrences in the order they were sent. Therefore, the event queue cannot be part of each event source. Finally, composed JEScala events make it possible to combine event occurrences with a different synchronicity, which needs to be supported by the event queue.

When an event only emits asynchronous occurrences without an argument, an observing disjunction can use a counter as an event queue for that event. Currently, the programmer can explicitly promise a disjunction that an observed event meets these requirements. We implement disjunctions in a class, this class provides the programmer with a method `markNullaryAsync` to specify that an event without arguments is always triggered asynchronously. A future extension could analyze the event graph or use another kind of static analysis to no longer depend on the programmer to call the aforementioned method. There is no order for such event occurrences, since they are indistinguishable by an argument, nor by a waiting thread. A runtime exception will be thrown after any invalid attempt to enqueue a synchronous event occurrence into such a specialized counter-based event queue.

7.4 FSM DSL

Collecting the transitions with the trait `FSM` The trait `FSM` provides the infrastructure that makes it possible to collect, at runtime, the transitions of the FSM, including an optional reaction. This uses two basic facilities provided by Scala: the possibility of using an infix operator notation for method calls and user-defined implicit conversions, called *implicits*.

The first point relates to the fact that s `->` t `on` a is just alternative syntax for s`.->(`t`).on(`a`)`. It may then look like `->(` should be a method that is part of some

type representing states, but the type of states S does not provide such a method. This is where the second point comes into play. The trait FSM defines an implicit conversion to wrap a source state into an instance of the class Arrow. An instance of the class Arrow stores the data collected for a transition. The class defines all the methods ->, on, triggers, and apply as setters of instance variables, destination state, action event, and reaction, respectively. These methods return this in order to be composed. After setting both the destination state and the action event via their respective setter method, the setter that was called last stores the transition in a temporary collection.

The trait FSM offers the method remove to wrap its argument, the source state, in an Eraser instance. This is similar to the Arrow instance except that it removes transitions from the temporary collection when the source and destination states and the action event are known.

The method initialState of the FSM trait sets a flag frozen to mark that no further instances of Arrow and Eraser can be created. It then turns the temporary mutable collection of transitions into an immutable collection of transitions. It also generates auxiliary sets like the set of states and the set of action events.

The method initialState is overriden by the implementation traits in order to finish the collection of transitions and start to build a state machine.

Join-based implementation The implementation trait FSM_J, which extends FSM, generates structures similar to those in Figure 5.8.

This means that the method generates first an imperative asynchronous event for each state, for instance the events Free and Busy in Figure 5.8. Then it defines a disjunction, with a binary Join pattern per transition. Such a Join pattern combines the state event and the action of the transition. If the transition defines a reaction, an additional handler is registered with the result event associated with the pattern. Next, it creates for each state with more than one incoming edge a declarative event. This event combines the resulting events from transitions to that state (e.g. the event toFree in Figure 5.8). When there is only a single incoming edge, the declarative event is only an alias for the result event. For each of these declarative events, a handler is registered with the event. This handler triggers the state event corresponding to the destination event. Finally, the initial state event is triggered.

Modifying an FSM by Inheritance To support subclasses that modify the description, the trait FSM offers the method unfreeze, which resets the information that was collected after calling initialState. The superclass already generated data structures to implement the FSM. These structures cannot always be modified. For ex-

ample, disjunctions cannot be extended after they have been created. However, these structures can be disabled and replaced by the modified version after the subclass calls `initialState`.

Optimizing Joins JEScala disjunctions use event queues for every event they observe. Our optimization only uses event queues for *action* events. The elements in the queue can contain a blocked thread, but remain empty for asynchronous events. For synchronous events they store the thread that delivered the event.

When an event occurrence arrives, it adds an element to its associated queue. The thread that delivered the event occurrence does not try to match any patterns, instead it tests whether the state machine has any *action* events waiting in the group of event queues that are associated with the valid events for the current state. When there is more than one event possible it selects one at random. We rely on the Scala implementation of the `random` function to ensure that all waiting events have a fair chance. The state machine finds the next state via a transition function that takes the current state and action event as its arguments. If the selected action event was synchronous, it unblocks the stored thread of the consumed *action* event.

A first optimization (`FSM_N`) uses a hash map as a transition function. The second optimization (`FSM_E`) assigns a sequence number to all used states and to the events. As a result, the transition function can be implemented as a two-dimensional array.

7.5 Event Monitors

Event monitors ensure that handlers belonging to the same user-defined group are not executed concurrently. For this, we rely on a component that we called a *synchronizer*. The synchronizer is related to actors. However, we mentioned in Section 6.2.2 several differences. We explain how we bridged these differences in an actor-based implementation of the synchronizer. Since Scala actors receive messages instead of JEScala events, we need two conversions: first, event occurrences of the observed events are converted to asynchronous actor messages; second, the actor reacts to these messages by emitting the parameter of the original event occurrence via a representing synchronous event. We explain first a simple situation. To reduce the complexity, we restrict the synchronizer to a single event of a fixed type. To implement this with an actor, we register a handler with the original event. The handler sends a message that wraps the argument of the event occurrences to the synchronizer. The synchronizer actor handles this message by explicitly triggering a synchronous event with the same type as the observed event.

However, a synchronizer without the restriction that we introduced before has to provide a *representing* event for each observed event, independently of the type of the argument in the observed event. This *representing* event is synchronous and has the same type of argument as the observed event. The restricted solution that we just explained cannot support different events since it needs a separate case class to represent a dedicated message for each observed event. Since the underlying Scala supports type parameters, it is possible to implement messages with a single case class that uses a type parameter. By using this mechanism, we can create a representing event for each observed event and send the parameter of the observed event via a message to the synchronizer. Then the actor needs to retrieve the type of the event parameter before triggering an event. However, the type is lost due to Scala type erasure. To solve this, a unique identifier is assigned to each observed event. Before the handler of the observed event sends the message to the synchronizer, the event stores this unique identifier in the message as well. The reaction of the synchronizer uses this identifier as a key in a map to retrieve a handler that typecasts the parameter that is stored in the message to the original type. Then it triggers the representing synchronous event that is created by the synchronizer with the correctly typed argument from the message.

Validation and Conclusion

Evaluation

Contents

8.1 Static Evaluation 152
8.2 Dynamic Evaluation of JEScala 154
8.2.1 Comparison with other languages 154
8.2.2 Effect of patterns complexity 155
8.2.3 Effect of optimizations 157
8.3 Qualitive Evaluation using a Ray Tracer 158
8.3.1 Conversion of the Ray Tracer 158
8.3.2 Principles of the Implementation of the Extensions 160
8.3.3 Conclusion 161
8.4 FSM DSL 162
8.4.1 Code size 162
8.4.2 Performance 163
8.5 Event Monitors 166

In this chapter, we evaluate three parts of our work:

— The language JEScala

— Our DSL for FSMs and the generated implementations.

— Adding concurrency to an EScala program using Event Monitors.

We demonstrate the design advantages of JEScala in several small case studies and provide a preliminary performance evaluation. The code used for the evaluation is available online [63].

Our DSL to describe FSMs is evaluated in two ways. First, we derive formulas to calculate the number of lines of a description with the DSL and the length of the generated Joins-based description. Then we use a benchmark to compare different implementations of the same state machine.

To evaluate the idea of the Event Monitor, we reuse parts from an existing EScala application to build a concurrent JEScala implementation. The target of this case study was reusing unchanged parts from a sequential program in a concurrent program. To evaluate the performance of the resulting concurrent version we removed the I/O bottlenecks in both the original version and the concurrent version.

8.1 Static Evaluation

Using several small case studies has two advantages over using a single larger one. First, with several *synthetic* examples we can challenge JEScala with intentionally complex coordination schemas, while a *real* application would probably be less compelling from a coordination standpoint. Second, a larger example would dilute the coordination schema within the application logic. On the contrary, our studies distill the essence of a coordination schema and sharpen the effects that we want to observe.

The case studies (Figure 8.1 first column) include classic concurrency patterns (e.g., critical section, producer-consumer, actors), simulations that require coordination across several components (e.g., cellular automaton, binary adder, virus spreading over a complex network), and the Web server running example from Chapters 3 and 4. Case studies also include client code that stresses the implemented features, e.g., threads accessing the critical section. The second and third columns in the figure report the number of threads and the number of components (classes and Scala objects) for each case study. We implemented each case study in JEScala and a subset JL (for Join Language) of JEScala excluding its specific features. JL programs are direct encodings of Polyphonic Scala programs (see Section 2.5.3 and Chapter 3) in JEScala.

For each implementation we measured the following metrics: lines of code ignoring comments and white spaces measured by CLOC [1], number of events, number of handlers and number of times imperative events are triggered. To test the effect of different

1. `http://cloc.sourceforge.net`

Case Study	Th.	Comp.	LOC		Events		Handlers		Imp.Evts	
			JL	JE	JL	JE	JL	JE	JL	JE
Critical Section (CS)	3	5	67	60	3	5	1	0	3	1
Alternating CS	3	4	49	42	7	8	5	1	8	3
Condition Variable	3	3	56	56	6	8	3	3	5	2
Monitor	6	7	86	80	9	10	3	1	7	2
Concurrent Barrier	2	3	46	37	4	4	1	0	3	0
Readers-writer Lock	6	7	81	71	12	8	8	3	12	3
Threadsafe Counter	5	5	47	44	6	6	3	1	6	4
Hoare Cond. Crit. Region	4	7	90	71	7	9	4	1	7	2
Rendezvous	2	3	68	64	3	3	1	1	2	0
Concurrent Futures	2	3	58	48	7	7	3	3	5	2
Producer-consumer (PC)	2	3	79	72	8	8	0	0	4	0
PC (Bounded Buffer)	4	4	72	68	7	7	2	2	5	3
Finite State Machine ST	1	2	76	66	14	11	11	5	12	6
Finite State Machine MT	4	4	74	64	14	11	11	5	12	6
Petri Net ST	1	2	46	44	9	12	13	14	9	8
Petri Net MT	3	2	56	54	11	14	12	11	9	8
Semaphore Petri Net	2	4	56	51	5	6	2	1	4	1
Tennis Players Petri Net	3	2	80	74	13	16	18	9	15	6
Agents (3 Ping-pong) ST	1	4	67	64	7	7	4	4	7	6
Agents (3 Ping-pong) MT	4	3	60	57	3	5	1	1	5	2
Agents (Token Ring) ST	1	4	54	54	7	6	4	4	5	4
Agents (Token Ring) MT	4	3	53	54	3	3	1	1	3	2
Elem. Cellular Automaton	1	3	87	84	8	11	4	1	9	2
Game Of Life	1	1	95	74	25	17	27	2	27	2
Shift Register	1	2	36	30	6	6	3	1	9	8
4 Bit Binary Adder	1	4	82	69	20	19	8	3	12	3
Logic Ports Circuit	3	4	70	64	10	7	7	1	7	1
Random Walks	7	8	185	179	16	16	12	3	17	4
Parall. Graph Explor.	21	20	184	181	9	10	5	6	10	7
Epidemic Model ST	1	11	184	172	18	16	16	4	18	6
Epidemic Model MT	11	11	190	178	18	16	16	4	18	6
Web Server	2	4	44	41	4	4	2	1	4	0
Web Server (Extended)	3	5	75	68	7	6	4	0	7	0
Web Server (Chapter 4)	4	5	111	97	18	18	8	4	16	8

Figure 8.1 – Main metrics for the case studies.

concurrency solutions, some case studies are implemented in both a single-threaded and a multithreaded version – marked with ST, respectively MT in the table. The need for coordination in a single-threaded context is not a contradiction, since a single thread must be "scheduled" to accomplish several tasks in a coordinated way. The variability between the ST and the MT versions does not significantly affect the following results.

Based on the numbers reported in Figure 8.1, we make the following observations. The implementations of Petri Nets and Parallel Graph Exploration use handlers to represent transitions, therefore we do not expect many differences between both implementations. JEScala captures coordination schemas in a more compact way. JEScala implementations have less lines of code (Columns 4–5). The proportion of event declarations required by JEScala and by JL depends on the case study (Columns 6–7). JEScala implementations define fewer handlers (Columns 8–9). Further, the number of imperative triggering of event is considerably reduced in the JEScala versions (Columns 10–11). The reduction of handlers and imperative events indicates that the coordination logic is moved from handlers and imperatively triggering events to declarative event expressions. As a consequence, developers do not risk forgetting firing events and coordination patterns are more composable, easier to extend, and express the programmer intention in a more declarative way.

8.2 Dynamic Evaluation of JEScala

The above evaluation addresses the advantages provided by JEScala when dealing with source code. To gain an idea about JEScala's performance, we implemented a number of benchmarks [2].

8.2.1 Comparison with other languages

We first compare JEScala with other languages that support Joins. The benchmark consists of an automaton with n states. All states have at most 2 neighbors, which can be visualized as an elevator in a building with n floors. A transition fires when the join of the event associated to the current state and the event associated to the transition to another state fires. We measured the throughput (i.e., joined events per second) for ScalaJoins, JEScala, the Esper complex event processing engine [35], JoCaml and Cω (Figure 8.2). Varying n from 1 to 5 did not change the results significantly, therefore

2. All measurements were performed on a MacBookPro6.2 with CPU I7 (2 cores, 2.66Ghz) with 8Gb ram, running OSX 10.6.8, Java 6 and Scala 2.10.

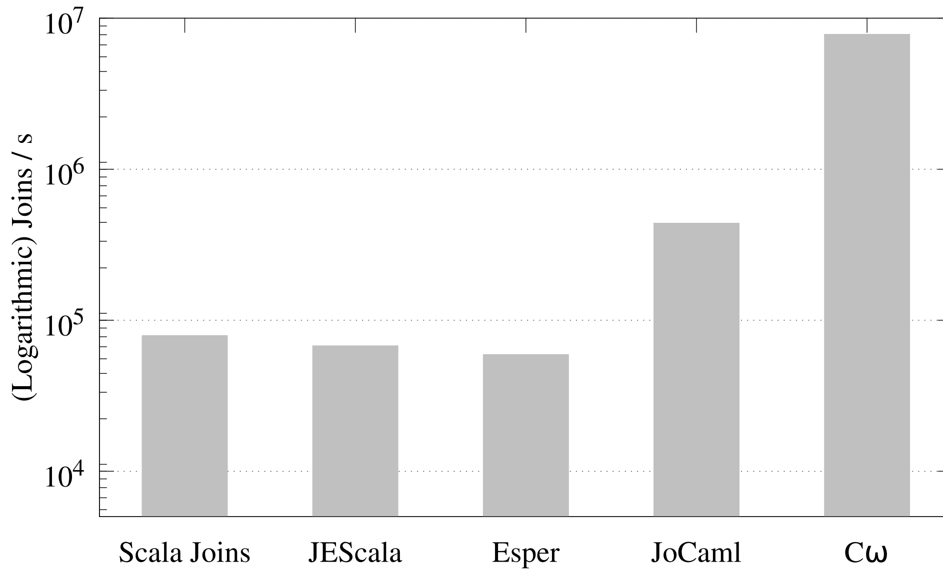


Figure 8.2 – Performance of Join languages.

we used the average of these 5 results. Note that a single state has only looping transitions. Languages based on a dedicated compiler like JoCaml and $C\omega$ have the best performance. The results also show that performances degrades when languages become increasingly expressive. This result is not surprising: for example $C\omega$ intentionally limits the constructs available to the programmer to achieve better performance [9]. At the other extreme of the spectrum, Esper supports an extremely expressive language for event composition. JEScala exhibits a level of performance that minimally outperforms ScalaJoins. However, JEScala is more expressive since it supports implicit invocation, event composition and real asynchronous events.

8.2.2 Effect of patterns complexity

The complexity of the matching patterns has an impact on performance that we measured with a dedicated benchmark. The size of the matching patterns in the benchmark increases; the cases $n = 3$ and $n = 4$ are shown in Figure 8.3a and Figure 8.3b. We measured the performance of each language for $n \in [2..6]$. The results for each language, normalized to the case $n = 2$, are shown in Figure 8.4. The benchmark shows a degradation of the performances when the size of the pattern increases. $C\omega$ and JoCaml outperform JEScala, which however does better than ScalaJoins. For JEScala, the first step (n from 2 to 3) reduces the performance to approximately 25%, further increasing n results in a nearly horizontal line in the graph.

```
1 var cnt:Long=0
2 imperative sync evt a[Unit]
3 imperative sync evt b[Unit]
4 imperative sync evt c[Unit]
5
6 evt (toC, toB, toA)=
7       ( a & b )
8     | ( c & a )
9     | ( b & c )
10
11 toA += (()=> { a(); cnt+=1 } )
12 toB += (()=> { b(); cnt+=1 } )
13 toC += (()=> { c(); cnt+=1 } )
```

(a)

```
1 var cnt:Long=0
2 imperative sync evt a[Unit]
3 imperative sync evt b[Unit]
4 imperative sync evt c[Unit]
5 imperative sync evt d[Unit]
6 evt (toD, toC, toB, toA)=
7       ( a & b & c)
8     | ( d & a & b)
9     | ( c & d & a)
10    | ( b & c & d)
11 toA += (()=> { a(); cnt+=1 } )
12 toB += (()=> { b(); cnt+=1 } )
13 toC += (()=> { c(); cnt+=1 } )
14 toD += (()=> { d(); cnt+=1 } )
```

(b)

Figure 8.3 – Benchmark: Increasing complexity of matching patterns with $n = 3$ (a) and $n = 4$ (b).

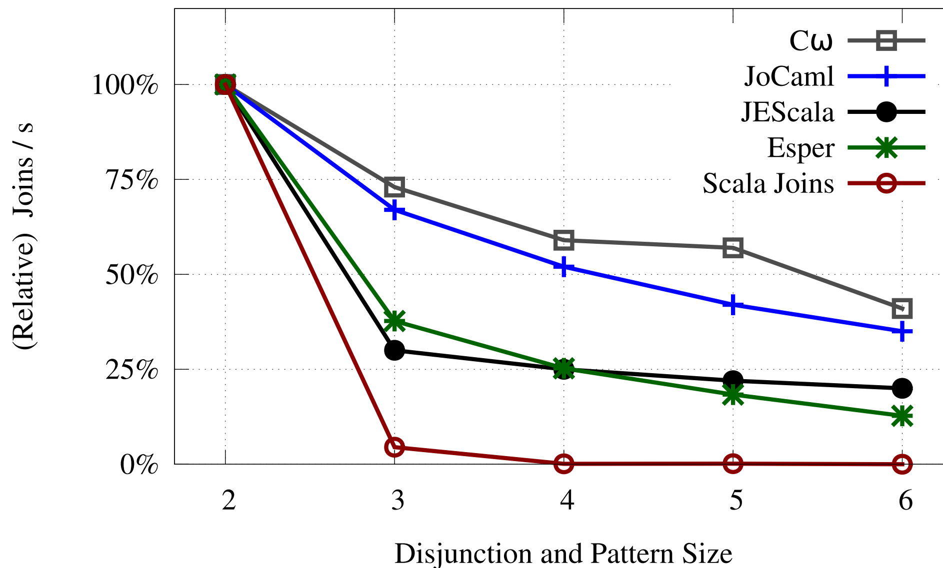


Figure 8.4 – Effect of growing pattern complexity on performance.

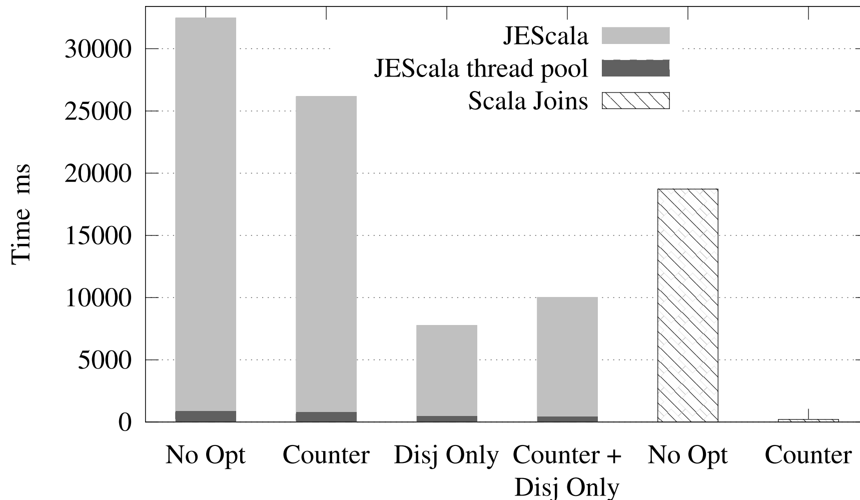


Figure 8.5 – Optimization of asynchronous events in JEScala.

8.2.3 Effect of optimizations

To measure the effect of the optimizations for asynchronous events described in Section 7.3.1, we implemented a simple version of the rock-paper-scissors game. Two players running in different threads trigger an event that corresponds to *rock*, *paper* or *scissors*. A game component matches those events in a disjunction. Each pattern in the disjunction captures a possible combination. Depending on the matching pattern, the first or the second player wins. We measured the time required to run $5 \cdot 10^4$ games.

The results can be seen in Figure 8.5. Column *No Opt.* shows the non optimized version of JEScala. Subsequent columns show the effect of the *Counter* optimization, of the *Disj. Only* optimization, and of both optimizations in action. To give an intuition of what the values mean in absolute terms, the last two columns show the performance of ScalaJoins in the same benchmark for events with and without parameters. The former is obtained by adding a dummy parameter to the event, instead of changing to the ScalaJoins *Counter* optimization, the latter shows the case in which the *Counter* optimization is applied.

The dark bars in Figure 8.5 show the performance of JEScala when adding the *Thread Pool* optimization. Figure 8.6 focuses on this case. The *Thread Pool* optimization is by far the most important to improve the performance of JEScala, and it is sufficient to make JEScala faster than ScalaJoins in the case of an event with a parameter (see columns *No Opt.*). However, other optimizations are also significant and further double the performance of JEScala.

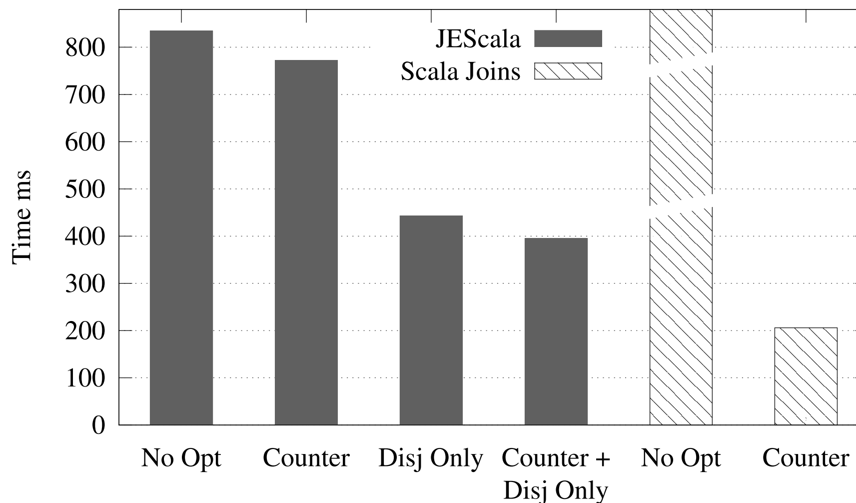


Figure 8.6 – Optimization of asynchronous events with the *Thread Pool* optimization.

8.3 Qualitive Evaluation using a Ray Tracer

To study the flexibility of JEScala, we started with an existing ray tracer written in JoCaml. This non-trivial application is described in [78]. For our experiments we manually converted twice a major part of this program to JEScala. We also considered three extensions to the ray tracer. A first extension shows the progress of the ray tracing by showing the image on the screen in real time. The second extension aims to speed up processing by saving intermediate results on disk. This makes it possible to create several pictures in a batch, without requiring interaction after each step. The third extension compares the results returned by multiple servers that received the same input through a *broker*. In case these servers try to save calculation costs by returning fake results the broker will not forward these results to a client. In Chapter 3, we used a synthetic language, Polyphonic Scala, to discuss the shortcomings of Join languages. However, now that we have described our language JEScala in Chapter 4, we are going to compare a subset of JEScala, close to what JoCaml offers and a full-blown version of JEScala.

8.3.1 Conversion of the Ray Tracer

In order to translate JoCaml into JEScala, we need to take into account the fact that both languages implement channels in a different way. In JoCaml, channels look like functions, which return values (this defines synchronous channels) or not (this defines asynchronous channels). In JEScala, the channels are a part of the event system, and

their receiver does not return anything to their sender. In this system the triggered primitive event defines the synchronicity of the event occurrence. JEScala also includes features (implicit invocation and declarative events) that have no direct counterparts in JoCaml. When only a single disjunction observes a JEScala event, it is possible to mimic JoCaml channels by explicitly defining the synchronicity of the event.

We created two JEScala versions by porting the JoCaml ray tracer twice, each using a different style. In the first version, we used a restricted JEScala to mimic JoCaml. This version does not rely on declarative events and it mimics static registration by not modifying the initial registration of handlers with events. This version mimics JoCaml by only using imperative events, which must be associated once and for all to a single handler or disjunction. This makes our code close to the design of the original JoCaml implementation, without requiring the reader to get familiar with yet another programming language. We refer to this version of the application as the simulated JoCaml version. The second version implements the ray tracer without restricting JEScala and it exploits features from implicit invocation and declarative events. We refer to this version as the full JEScala version.

The original implementation of the JoCaml language supports distribution. However, the JoCaml developers created a new implementation, which no longer supports distribution. JEScala does not support distribution over multiple hosts either. Therefore, we implemented the ray tracer in a single virtual machine without any distribution. Implementing the program with the new version of the JoCaml language could apply the same solution. Both JEScala implementations still model a client-server interaction by implementing the client and the server as separate modules that directly interact instead of using a network.

The original JoCaml implementation models classes as functions that construct a tuple of method-like functions. In both JEScala versions, we use classes and objects. We did not mimic the functional JoCaml implementation of an iteration in JEScala but rather used the sophisticated `for` expressions of Scala. This required an alternative to the JoCaml pattern that concurrently applies a function over an enumeration of values representing the lines in a bitmap image. In JEScala, we can trigger asynchronous events in a loop to implement the concurrent ray tracing of multiple lines in an image. The full JEScala and the simulated JoCaml implementations are simpler than the original JoCaml one, because we pass the reference to a shared array to represent the image as part of the event. Since each event occurrence writes its result in a dedicated index of the array, there are no race conditions. We did not implement all features from the JoCaml version. For instance, the JoCaml implementation interprets a description of a three-dimensional scene, we used for our experiment a fixed scene. We also omitted the saving of the resulting image and only display the resulting image after completing its calculation.

A ray tracer creates two-dimensional images from a three-dimensional scene, by tracking an imaginary light ray back from the eye of the observer through a two-dimensional screen to the objects of a three-dimensional scene. It is possible to consider reflection and other optical phenomena in the calculation of the color.

Our ray tracer concurrently calculates lines in the two-dimensional image. Choosing a finer-grained concurrency, for example concurrently calculating single pixels, would cause many asynchronous event occurrences, which would all require interaction among threads. The resulting overhead would overcome the gain brought by concurrency.

The objects in the three-dimensional scene are represented by instances of subclasses of `WorldObject`. Similar to the JoCaml ray tracer, only spheres and planes are supported. Conceptually, a `WorldObject` models a material with reflection and exposes a method to check whether it intersects with a given ray. A ray is only affected by a `WorldObject` when it intersects with it. This makes it possible to avoid calculations for pixels that are the result of rays that do not intersect with anything and appear black.

In terms of client-server interactions, the clients construct a two-dimensional image by requesting the calculation of image lines from a server that stores a three-dimensional scene. The entire scene can be turned around two axes in the ray tracer. The two axes are named like the motions of an airplane: *pitch* and *yaw*. Pitch means turning an object around the horizontal perpendicular axis to the eye, while yaw turns the scene around the vertical axis. A third motion, *roll*, which is not implemented, would turn the resulting image around a third axis that points to the eye of the observer.

8.3.2 Principles of the Implementation of the Extensions

We consider two alternatives: extending the simulated JoCaml version using simulated JoCaml or extending the full JEScala version with full JEScala. In the simulated JoCaml version, we need to insert calls to the code of the extensions, these modifications can introduce mistakes and also make future changes more complex. In the full JEScala version, we have two options to implement these extensions: defining new implicit events or using the already-defined events (by composing them or defining new handlers).

In the first extension, we need to extract each line after it is completed. In the simulated JoCaml implementation, we inserted code to explicitly emit each calculated line, because implicit invocation is not available. In the full JEScala version, we can use implicit invocation and register a new handler with the event `lineRendered`, which is triggered each time a line of the image is completed. Since the number of lines in the image is known, we can observe the progress and show the constructed image. This extension increases the total execution time compared to the version that only computes the image.

The second extension collects all the lines and writes the image on the disk only once the complete picture is available. The resulting images can easily be combined into a movie, by using a tool, like ffmpeg [100]. In order to generate several images of the same scene with different point of views, we directly call the appropriate methods with the values defining the point of view. In the simulated JoCaml version we need to change the code that writes the lines. To write image files in the full JEScala version, we collect the image line-by-line by observing the same event `lineRendered` as for the first extension. This extension might not look exciting in the first place. However, several image files can be created this way.

The third extension is inspired by projects that distribute packages with independent calculation tasks over many systems (e.g., SETI@home). It relies on volunteers to contribute calculation time to a (supposedly) distributed ray tracer system. People are motivated to participate in ray tracing a movie by offering some reward. However, when ray tracing happens on servers that run on computers that we cannot control, we need to make sure that these servers return correct results. In other words, we need to prevent people from cheating by collecting rewards for sending incorrect data.

To implement this third extension, we used the existing part of a client that requests a server to ray trace a three-dimensional scene. For this experiment we rearrange the collaboration between a single client and a single server by introducing a *broker*, to which the client sends its requests. The broker acts as a server for the client and as a client for the server. The broker sends requests to multiple servers and only forward results that were verified to match from more than one server. We had to modify both client and server in the simulated JoCaml version instead of reusing existing components. The lack of dynamic registration in simulated JoCaml requires an explicitly managed set of servers that is part of the broker. In the JEScala version events are used for the communication between a client and a server. The client emits requests to which the server reacts by triggering an event with the result of its calculation. The set of servers managed by a broker with JEScala dynamic registration can change.

8.3.3 Conclusion

We converted an existing ray tracer written in JoCaml to JEScala and considered three extensions to this ray tracer. We did so using a subset of JEScala mimicking JoCaml, without implicit invocation and implicit events as well as using a full-blown version of JEScala. In the first case, implementing the extensions required invasive modifications of the ray tracer, whereas the advanced features of JEScala made it easy to implement the three extensions without any modification. Implicit invocation makes it possible to use existing events while implicit events make it possible to introduce new events of interest to the extensions.

8.4 FSM DSL

There are two directions to evaluate our DSL for Finite State Machines (Section 5.4). First, we measure the number of lines of code (LOC) that can be saved with our DSL compared to a complete implementation of an FSM. Second, we evaluate the performance of the different implementations and we also measure the performance of the described actor and state pattern based solutions.

8.4.1 Code size

We express the LOCs required to define a FSM as a function of the number of used states and the number of transitions in the FSM. For an FSM without reactions, we show the LOCs for our DSL and for an equivalent implementation based on Joins.

As can be seen by looking at Figure 5.10 and generalizing it, the number of lines of the description of an FSM using our DSL depends on the number T of transitions and the number S of states. It consists of:

1 line defining the type of the FSM states.

S lines, one per state defining the corresponding object.

1 line for the header of the class declaration that implements the FSM.

T lines, 1 per transition.

1 line for the initial state.

1 line to close the class declaration.

$$\begin{aligned} LOC &= 1 + S + 1 + T + 1 + 1 \\ &= 4 + S + T \qquad \text{(DSL description)} \end{aligned}$$

The number of lines of an implementation based on Joins can be computed by looking at Figure 5.8 and generalizing it. Here, a complication comes from the fact that a state with several incoming edges (`Free` in the example) requires an additional definition of a declarative event (`toFree`), whereas the corresponding event (`toBusy`) is rather declared together with the disjunction if the state (`Busy`) has a single incoming edge. For the sake of simplicity, we only consider the extreme cases: no state requires an additional event definition or every state requires one.

This gives:

1 line to define the class that implements the FSM.

S lines, one per private asynchronous state event.

1 line to declare the events resulting from the join patterns.

T lines, one per transition implemented as a join pattern.

between S lines (worst case) and none, one per event combining several action events.

S lines, one per state, to define a handler triggering the corresponding state event.

1 line for the initial state .

1 line to close the class declaration.

This gives in the worst case:

$$\begin{aligned} LOC &= 1 + S + 1 + T + S + S + 1 + 1 \\ &= 4 + 3 * S + T \end{aligned} \tag{Joins Implementation}$$

and in the best case:

$$LOC = 4 + 2 * S + T$$

The formulas for DSL description and Joins Implementation show that using the DSL for defining an FSM reduces the number of lines between once and twice the number of states. However, this result does not show the complexity of the Joins-based implementation hidden from the programmers by the DSL. An additional transition introduces an extra Join pattern with an extra resulting event. This event needs to be added to the declarative event for the target state. Any optimization to the Joins based implementation can make this more complicated. While we do not provide numbers for the complexity that such changes cause, modifications in the DSL description are less cumbersome.

8.4.2 Performance

To evaluate the performance of our solution, we implemented the triangular finite state machine from Figure 8.7 with our DSL (Figure 8.8). Each state connects with both others via transitions by the actions `cw` (clockwise) and `ccw` (counterclockwise). In the resulting graph, we can move from each corner in two directions: clockwise and counterclockwise. The class `TriangleBench` extends the trait `FSM` that we replace

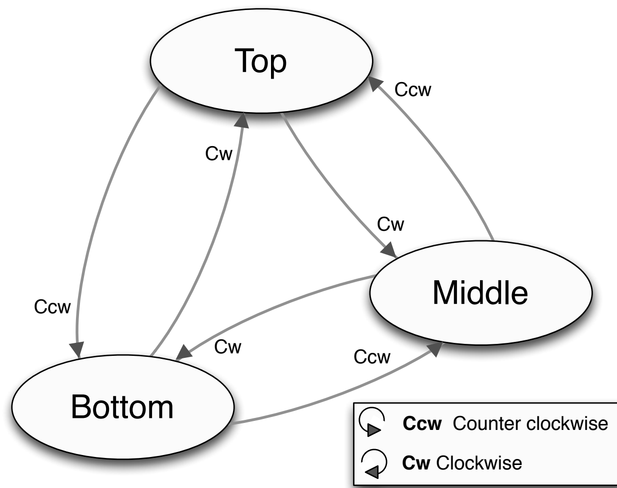


Figure 8.7 – FSM for the triangle benchmark.

```
1 trait Corner
2 object top extends Corner
3 object bottom extends Corner
4 object middle extends Corner
5 class TriangleBench(val n:Int, val c:Client) extends FSM[Corner] {
6   var counter:Int=0
7   def count= (()=> { counter+= 1 })
8    top -> middle     on c.cw    apply(count)
9    middle -> top     on c.ccw   apply(count)
10   middle -> bottom  on c.cw    apply(count)
11   bottom -> middle  on c.ccw   apply(count)
12   bottom -> top     on c.cw    apply(count)
13   top -> bottom     on c.ccw   apply(count)
14   initialState(top)
15 }
```

Figure 8.8 – Implementation of the triangle benchmark.

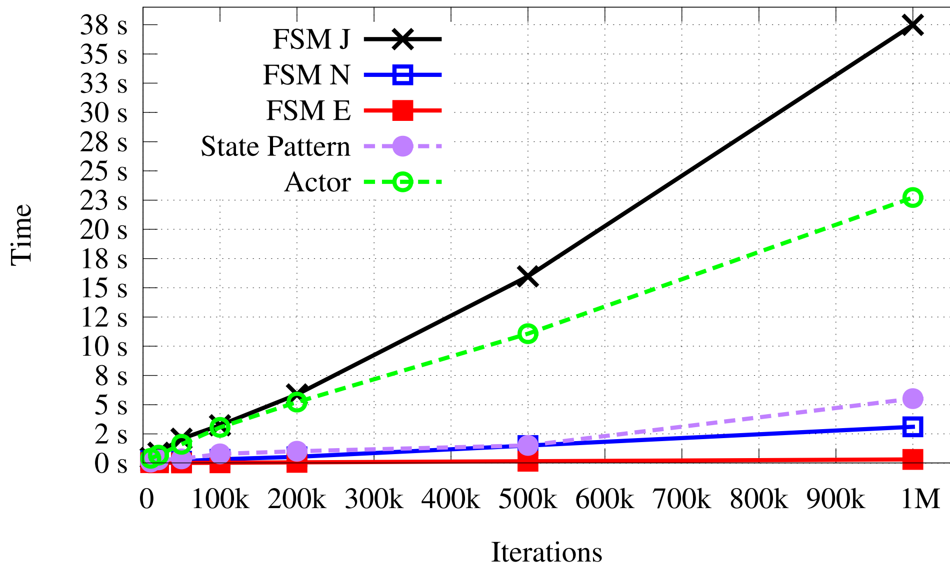


Figure 8.9 – Execution time for the triangle benchmark.

with the three different implementations for the benchmark. The client of this FSM uses a pseudo random generator [61] based on the Fibonacci numbers to reproduce the same sequence of `cw` and `ccw` events for each implementation. Each transition calls the method `count`, which stops the benchmark after reaching an increasing number of iterations. This benchmark (Figure 8.9) shows the time for the transition counter to reach a growing number (1 up to a million). Using an implementation that replaces part of the implementation of Joins by a `HashMap` instead of the original Join-based DSL does not add any complexity for the programmer.

The original implementation based on Joins (`FSM_J`) takes more than half a minute (37s) to handle a million action events. We had to increase the heap size for the JVM to run this benchmark with that amount of iterations. Section 5.4.2 describes two optimizations. The first optimization (`FSM_N`), which is based on a specialized replacement of a disjunction, divides the total time almost by 10 (3s). The second optimization (`FSM_E`), which also optimizes the transition functions by a two-dimensional table, increases the speed even more, reducing the time for a million iterations to 0.3s. Since the last number is small we can expect it to be only an indication that the last version is an improvement.

We compare these three implementations with two manual implementations of the triangle benchmark. First we implement the triangle with state pattern, monitors and events (Figure 5.5). Second, we use actors (Figure 5.4). We see that the optimized versions generated by our DSL are faster than the manual implementations.

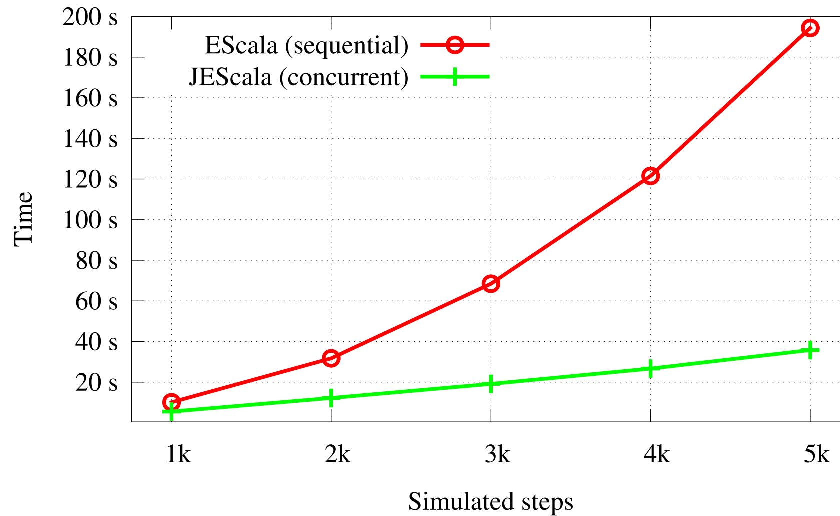


Figure 8.10 – Sequential and concurrent universe simulation.

8.5 Event Monitors

We started with an improved event-based version of the universe simulation described in [44]. We modified this implementation to end after a fixed number of cycles and we removed the code that shows the evolution on the screen. By not showing the progress on the screen we avoid the time-consuming output. Similar to the benchmark for FSMs in Section 8.4.2, we made sure that the simulations are repeatable, therefore we replaced in both implementations the Scala random generator with a Fibonacci based pseudo random generator and start with a fixed seed value. However, this pseudo random generator does not imply the same behavior for a sequential version and a concurrent version because in a concurrent version the interleaving of threads can result in non-determinism.

We use asynchronous events to make the program concurrent. A `Synchronizer` protects the access to the shared board. Chapter 6 describes *event monitors* as a way to ensure mutual-exclusion in the handling of events. The `Synchronizer` implements this approach. When we run this program, the elements on the board do their calculations concurrently after each clock tick.

We measure the time for an increasing number of simulation steps in a sequential event-based version and for a concurrent version that reuses parts from this sequential version. Figure 8.9 shows that the time for the event-based version reaches almost 200 seconds for 5000 iterations while the concurrent version needs less than 40 seconds.

The speed-up would probably be higher if we had not used the `Synchronizer` but had implemented the program from the start to work concurrently. We did not test this.

The synchronizer makes it possible to combine existing sequential parts of code, which interact with events. However, when the code relies on the blocking semantics of synchronously handled events, executing handlers asynchronously would require additional synchronization. The handlers can delay another component that waits for the effect of the execution of that handler. In our case study, handlers are not crucial for the progress of other handlers, therefore we did not study how to deal with such handlers.

Related work

Contents

9.1 Join languages 170
9.2 Calculi 172
9.3 Other languages 172
9.4 Other approaches 173

JEScala combines an expressive event system with elements from Join languages. This combination can express coordination patterns without modifying existing code by using Aspect-Oriented features that are an intrinsic part of the event system.

We compare JEScala with the languages that we discussed in Section 2.5.2, using a set of features. We mention major differences from JEScala as well. We mention related calculi, since these can help to describe our language with a formal model. Such a formal model can help to further refine JEScala.

Finally, we discuss other languages that aim to describe the coordination patterns among concurrent components.

	Language	Channels	Sync	Disjunction	Matching
JoCaml	Caml	Function	Both	Explicit	Non deterministic
Funnel	Funnel	Function	Both	Explicit	Non deterministic
Polyphonic C#	C#	Method	Both	Object	Non deterministic
Join Java	Java	Method	Both	Object	Both
ScalaJoins	Scala	Imper. Event	Both	Explicit	Non deterministic.
JErlang	Erlang	Message	Async	Actor	Deterministic
JCThorn	Thorn	Message	Async	Component	Deterministic
Join Conc. Lib.	.NET	Channel	Both	Explicit	Non deterministic
Concurrent Basic	Visual Basic	Channel	Both	Object	Non deterministic
Mogemoge	Mogemoge	Join token	Async	Global	Deterministic
JEScala	EScala	Adv. Events	Both	Explicit	Non deterministic

Figure 9.1 – Languages implementing Join abstractions.

9.1 Join languages

The table in Figure 9.1 summarizes key design aspects of Join languages. It completes the information shown before in the table from Figure 2.33. Section 2.5.2 gives an overview of Join languages, therefore we only mention the differences from JEScala. Most Join languages are based on existing idioms that are shown in the *Language* column. The *Channels* column shows how channels are implemented. The *Sync* column indicates whether channels are synchronous or asynchronous. The *Disjunction* column shows how disjunctions are defined: through a specific explicit construct (*Explicit*) or, implicitly, via existing language abstractions. The *Matching* column shows how to select a reaction among active ones.

What sets JEScala truly apart from other Join languages is its advanced event system that supports event composition and implicit events in addition to imperatively triggered events. To the best of our knowledge, JEScala is the only language that explores the synergy between such an event system and join operators. Chapter 3 discusses the effect of using imperative events rather than method or function calls to implement channels. The advantages of the synergy of implicit and declarative events with Joins were discussed in Chapter 4.

The closest cousin of JEScala is the Scala library Scala Joins [51], since it also implements channels as events. Similarly to JEScala, disjunctions are explicit constructions. However, a class can only contain a single disjunction. The events system of Scala Joins is simpler than the one of JEScala, composing events is not possible. A Join patterns can only observe the events that are completely defined in the same object as the patterns. The complete definition of an event includes the type of its parameter and

its synchronicity. When considering ScalaJoins as a building block for implementing JEScala, the necessity of predefining the synchronicity of the events provided by the library turned out to be problematic for implementing JEScala declarative events, whose synchronicity may vary from one occurrence to the other.

Unlike JEScala, Polyphonic C# [9] supports only one synchronous method per pattern. Subclasses can change the body of a reaction but cannot extend disjunctions with new reactions. For our experiments we used the research language Cω [106], which offers the same extensions to C#.

The Join Concurrency Library [108] and its successor [122] have the same features as Polyphonic C#. The same library has been integrated into Visual Basic, this resulted in the language Concurrent Basic [109]. This language has syntax to define channels. A function can contain a `when` clause after its header. This clause begins with the keyword `when` followed by a comma separated list of channels to define a Join pattern. In addition to what the library supports, subclasses in Concurrent Basic can add new patterns to the disjunction of their super class, as proposed in the Objective Join Calculus [40]. However, this incurs issues that are discussed in [9]. To avoid such issues Join Java [60] prevents inheritance on classes that act like a disjunction. In this languages, the programmer can decide on the strategy to choose the matching pattern when an arriving event can match more than one: First, this choice can be non deterministic as in the chemical model; Second, it can choose the first pattern in the order they are defined can be chosen.

Funnel [90] is, as JoCaml, a language that explicitly uses the Join Calculus as its foundation (with the variation that only one synchronous channel is allowed in a Join pattern).

In JErlang [97], channels are messages exchanged by actors. Erlang patterns are extended to express matching of multiple subsequent messages. Patterns are matched in their declaration order.

JCThorn [94] extends the scripting language Thorn [12]. *Components* are actor-like containers for objects that share the same mailbox. JEScala implements a finer-grained event system and, unlike in JCThorn, concurrency abstractions and events are at the object granularity level.

Mogemoge [87] is a prototype-based scripting language for game programming. Unlike the other Join languages that we discussed before, it has only one single global disjunction. This language defines interactions among game characters are by using asynchronous events called *join tokens* which share a single global disjunction, the *token pool*.

9.2 Calculi

JEScala has been designed in a pragmatic way. Proper theoretical foundations for JEScala could result into a better understanding of the properties of this language.

In Section 2.5.1.2 we explained the Join calculus, which is the formal foundation for the Join languages that we discussed. Inplicit invocation was formally studied by Garlan and Notkin [43], further the Aspect Join calculus [117] may also be a source of inspiration with respect to implicit invocation.

Different from other Join languages, the synchronicity of events in a Join pattern can vary. The effects of varying synchronicity for declarative events might be better understood with a formal model.

A formal model also could help to refine the design choices that we made. For example, such a model can help to explore the combination of JEScala's eventsystem with inheritance that can modify disjunctions as studied in the objective Join calculus [40]. From the Join languages that we studied, only Concurrent Basic (Section 2.5.2) can change a disjunction via inheritance.

9.3 Other languages

Pāñini [75] is a programming language that aims at coordinating concurrent components in a program by using explicit typed events, à la Ptolemy [103], except that these events are asynchronous, with a different meaning than the one we have used so far: events are fired synchronously with respect to their source but theirs handlers are executed asynchronously. However, no declarative ways of combining events are provided. Current versions of Pāñini aim at implicit concurrency [101, 102] with a programming style that is close to sequential programming. Using *capsules* to group objects into single threaded entities, which are combined into concurrent *systems*, results in coarse-grained concurrency comparable to actors [56]. Communication between capsules looks like method calls in a sequential program instead of asynchronous messages between actors.

Implicit invocation with traits [95] is another proposal based on explicit event types. There are no means of composition. The synchronicity of an event simply depends on whether it is associated to a block of code. In the first case, two synchronous events are defined, which are triggered *before* and *after* the execution of the block of code, otherwise a single asynchronous event is defined.

Join point interfaces [15] provide an interesting alternative to the event model of JEScala by using event types and still providing both explicit and implicit events, including

declarative events. This proposal is closer to Aspect-Oriented Programming than JEScala. Its events (join points) can return a value and its handlers (advices) can be composed with `proceed`. However, it does not include any specific support for concurrency.

9.4 Other approaches

Section 2.2.4 introduced Complex Event Processing (CEP), which correlates time-changing streams of data. The available operators include Joins with rich (and often subtle) semantic alternatives, which are typically applied to time windows. To evaluate the performance of JEScala, we chose Esper [35] to represent an CEP system. There are two reasons for this choice. First, Esper is freely available. Second, similar to JEScala it runs on the JVM. While Esper does not have native disjunctions, it provides the user with windows on event streams and a way to explicitly insert events in such window. The user can simulate disjunctions by combining these constructs.

Sequential and Parallel Object Monitors (SOM and POM) [21, 20] aim to separate fine-grained synchronization concerns from the application logic. An *object monitor* intercepts method calls and reifies them as objects, then it acts as a programmable threadless scheduler. The scheduler applies these reified calls to the methods that belong to the one or more objects to which the monitor is attached. It is a very expressive framework that can implement monitors in Java, that are able to deal with, for instance, Join patterns and disjunctions. In addition, it can implement different semantics, for instance in terms of determinancy. While *object monitor*s are programmable, our *Events-Monitors* (Chapter 6) only schedule the handling of event occurrences using a FIFO strategy.

JEScala also deals with separation of concerns and concurrency but not at the same level. Unlike the other approaches, our abstractions are fixed. Our interest is to improve modularity by seamlessly integrating these abstractions at the language level. Sometimes, but not always, both approaches almost completely overlap. For instance, in Java, it would make sense to write the rate limiter of Figure 3.4 as an object monitor. However, such implementation would require more programming work. When this rate limiter becomes more complex, its mapping to a monitor becomes unclear. Because the concurrency and the synchronization concerns cannot be defined in a separate module.

The work on Events and Composition [14] introduces a rich programming model combining a form of implicit event types and aspects. Two salient features of the proposal are the possibility to define, within event declarations, side-effect-free code collecting data to be carried by the event as well as fine-grained means to control composition. Concurrency issues are not addressed.

Summary and Future Work

Contents

10.1 Summary 175
10.2 Future work 176

10.1 Summary

This work has presented the design of JEScala, a language that combines the advanced event system of EScala with concurrency abstractions from the Join Calculus. We have shown that this solution captures coordination patterns in a more compact and more declarative way than existing Join languages, while preserving the OO style of modular reasoning, since events are object members. Still, we have found that concurrency issues related to non-determinism and the mixing of synchronous and asynchronous events (a source of deadlocks) is challenging. Future research can help the programmer to choose the synchronicity of all events, which currently is not always clear.

Because our case studies indicate Finite State Machines (FSM) as a recurring pattern, we have designed a domain specific language (DSL) to describe state machines that are driven by action events without arguments. Based on this DSL we explored dedicated

implementations of a join-like construct for FSMs, instead of using the general purpose Joins. This alternative implementation makes it possible to enhance performance. We show two examples of state machines that our DSL does not support. This limitation is the result of intentionally keeping our DSL simple by not supporting arguments for the action events nor for the state events. The first example originates from functional reactive programming (FRP), the second implements a reader-writer lock with a state machine without an upper limit for the number of concurrent readers.

10.2 Future work

Below we describe the future work on JEScala that we divide in the following main elements:

— Exploring the theoretical underpinning of JEScala.

— Improving performance.

— Providing compiler support.

— Exploring integrating time windows from CEP

— Exploring whether events with return types are possible.

— Exploring JEScala in other applications, e.g. reactive programming

A theoretical model for this language can help to understand effects of:

— The order that handlers are executed.

— Provide the programmer with guidelines to choose the synchronicity of events.

— Extending inheritance to Join patters as suggested by Fournet et al. [40]

While we do not have a detailed list, we consider to study the following ways to enhance performance:

— Caching event handlers when analysis shows that an event graph is not likely to change, which can be possible for parts of the event graph without filter.

— Convert events that are always synchronous and have a single fixed handler to a method call.

— Adapting approaches mentioned in [122] to the JEScala disjunctions that support events with a varying synchronicity.

During the experiments we directly used a library without any syntactic sugar. The syntax presented in Chapter 4 can be supported by modifying the compiler similar to what was done for EScala [44] or by using SugarScala, which is a recent Scala equivalent of SugarJ [34].

As we already discussed in Chapter 9, CEP engines offer richer semantics for event correlation than event-based languages – most noticeably, by including *time* in various types of windows over the event streams. We plan to explore the semantic alternatives that Joins offer in the context of event correlation over time windows. This field has been partially explored in CEP, but language integration of flexible semantics for correlating events is still a research challenge.

By extending events with return values the *proceed* construct is possible. However, the effect of such events on inheritance and other language features has to be studied.

The application of JEScala can be explored further, for example:

— Further experiments could result into a design for a DSL that can describe alternative state machines.

— Events monitors can help use parts for a sequential program to build a concurrent program. However, in our implementation the programmer is responsible for respecting the model, similar to simulating actors in an Object-Oriented language. We can explore whether it is possible to validate programs before running them.

— We can also explore JEScala for reactive programming [24]. In reactive programming, the order of calculations is a consequence of the availability of the arguments, since a later known argument causes a recalculation that changes its result, known as a *glitch*. In a sequential setting, a common approach is to use a central coordinator to order the calculations. However, in a concurrent setting the order can vary as the time for a calculation can depend on arguments. By using JEScala distributed coordination can be more flexible and furthermore it can be a step towards distributed coordination.

Appendix

Curriculum Vitae

7/2010 – 3/2015	Ph.D. student in two teams: ASCOLA (École des Mines de Nantes, fr) and the Software Technology Group (Technische Universität Darmstadt, de)
2001 – 5/2010	Professional experience in the companies: Inno.com, Transwide, Gudrun Informatics, Siemens/Egemin+ and Able/Vasco
10/1999 – 9/2000	European Master in Object-, component-, and aspect-, Oriented Software Engineering (EMOOSE) ($\approx$ Master after Master) *Vrije Universiteit Brussel (be), École des Mines de Nantes (fr) and Universidad de Chile (cl)*
10/1995 – 7/1999	Licentiaat in de Informatica ($\approx$ Master of Science) *Vrije Universiteit Brussel (be)*
10/1995 – 7/1996	Kandidaat in de Informatica ($\approx$ Academic Bachelor) *Vrije Universiteit Brussel (be)*
9/1992 – 6/1995	Graduaat in de Toegepaste Informatica ($\approx$ Professional Bachelor) *Hoger Technisch Instituut v/h GemeenschapsOnderwijs (HTIGO) Deurne (be)* (merged into Artesis Plantijn Hogeschool Antwerpen)

Zusammenfassung

Im Softwarekontext wird die parallele Ausführung von voneinander unabhängigen Aufgaben als Nebenläufigkeit bezeichnet. Die differenzierteste Art, Nebenläufigkeit zu erreichen, erfolgt über die Verwendung von Threads. Jeder Thread umfasst eine Sequenz von Anweisungen, die ausgeführt werden können. Während Nebenläufigkeit auch zwischen Programmen innerhalb eines Betriebssystems existiert, diskutiert die vorliegende Arbeit die Nebenläufigkeit innerhalb von Nutzeranwendungen – insbesondere Threads. Sobald einige dieser Threads für die Erfüllung einer Aufgabe zusammenarbeiten, müssen diese miteinander synchronisiert werden, um *race conditions* [86] zu vermeiden. Trotz umfangreicher Forschung und der Verwendung von Nebenläufigkeit, bleibt die Synchronisation von Threads eine komplexe Herausforderung.
Die Low-Level *Semaphoren* [29] sowie die abstrakteren *Monitors* [52, 58] sind grundlegende Synchronisationsmechanismen. Als Alternative gilt der Join Calculus [39], der *Join-Patterns* und Kombinationen dieser Patterns (*Disjunktionen*) als Schlüsselkonzepte einführt. Diese beschreiben die Interaktionen zwischen verschiedenen Prozessen, die über das Senden von Daten in Kommunikationskanälen miteinander kommunizieren. Im ursprünglichen Modell verläuft die Kommunikation asynchron. Trotzdem kann auf der Grundlage einer asynchronen eine synchrone Kommunikation aufgebaut werden.
Die Interaktion zwischen dem ausführenden Programm und ein oder mehreren Join Patterns ist nur möglich, wenn an den richtigen Stellen der Anwendung explizite Übergabepunkte definiert werden. Diese Übergabepunkte senden Daten über einen Kommunikationskanal zu einer Disjunktion. Wenn Methoden oder Funktionen solch einen Kanal modellieren, ist auch der Empfänger des Kanals durch Übergabepunkt festgelegt. Da diese Übergabepunkte auch mit dem Rest des Quellcodes verwoben sind, entste-

hen beim Senden von Daten implizite Abhängigkeiten – auch zwischen Join Patterns. Solche Abhängigkeiten verhindern modulares Design.
Die Programmiersprache, die in der vorliegenden Arbeit vorstellt wird, löst das Koordinationsproblem von nebenläufigen Programmen mithilfe von State-of-the-Art Programmiersprachen. Dafür wird die Synergie zwischen dem Konzept des Join Calculus und dem fortgeschrittenen Event System von EScala [44] verwendet. Das ursprüngliche Event System von EScala unterstützt zwei unterschiedliche Gruppen von Events: *primitive* und *deklarative*. Primitive Events teilen sich wiederum in *imperative* Events, welche explizit durch Anweisungen im Quellcode ausgelöst werden müssen, und *implizite* Events, welche ausgelöst werden, sobald die Ausführung des Quellcodes einen bestimmten Punkt erreicht. Implizite Events sind referenzierbare Teile des Quellcodes, welchen *Join-Points* in AOP [68] ähneln. *Deklarative*Events nutzen Eventausdrücke, um Events zusammenzusetzen. Eventausdrücke können Primitive, beispielsweise `before(m)`, enthalten, die sich auf das implizite Event beziehen, welches ausgelöst wird, bevor die Methode `m` zur Ausführung kommt.
Der übliche Weg, Interaktionen zu beschreiben, erfolgt über einen Automaten. Die vorliegende Arbeit stellt eine DSL vor, mit der Automaten beschrieben werden können. Bei der Verwendung unserer DSL wird der Quellcode durch die Beschreibung eines Automaten versteckt, der zur Implementierung geschrieben werden müsste. Basierend auf dieser DSL unterstützen wir unterschiedliche Implementierungen. Die erste Implementierung basiert auf Joins. Alle anderen nutzen performantere Alternativen, während die Semantik der Join-basierten Variante erhalten bleibt.
Event Handler können solange auf einen geteilten Zustand zugreifen, bis ein dezidiertes Verhalten erzwungen wird. ähnlich zum Objekt-Monitor [58, 52] verhindert auch unser Event-Monitor das parallele Bearbeiten von Events. Durch die Verwendung von Event-Monitoren können dadurch Abschnitte aus sequenziellen Programmen in parallelisierten JEScala-Programmen wiederverwendet werden.
Abschließend wird JEScala mit anderen Join-Sprachen anhand der Programmkomplexität sowie der Performanz verglichen. Weiterhin evaluieren wir die DSL, indem wir die explizite Beschreibung eines Automaten mit der Join-basierten Implementierung vergleichen. Zusätzlich vergleichen wir die Performanz der verschiedenen Implementierungen. Anhand der Verwendung von Event-Monitoren erstellen wir eine parallelisierte Anwendung durch die Wiederverwendung von sequenziellem Quellcode. Außerdem haben wir eine Strahlenverfolgungs-Anwendung von JoCaml zu JEScala portiert. Abschließend verwendeten wir das Event System von JEScala, um die Anwendung zu erweitern, ohne ihre Komponenten zu modifizieren.

Résumé en Français

Contents

C.1 Introduction **186**
C.1.1 Concurrence 186
C.1.2 Supporter la concurrence dans un langage de programmation . 188
C.1.3 La conception du langage JEScala 191
C.1.4 Résumé de la contribution 192
C.2 L'état de l'art **193**
C.2.1 Programmation par aspects 193
C.2.2 Programmation par événements 194
C.2.3 La combinaison d'événements et d'aspects 195
C.2.4 EScala 195
C.2.5 La concurrence 196
C.3 La problématique **199**
C.4 Un système d'événements avancé à la rescousse **202**
C.4.1 Événements impératifs 204
C.4.2 Événements implicites 204
C.4.3 Événements déclaratifs 204
C.4.4 Événements abstraits 205
C.4.5 Join patterns avec des événements implicites et déclaratifs . . 205
C.4.6 L'enregistrement dynamique 206
C.5 Automate fini **207**

C.5.1 Programmation par objets 207
C.5.2 Programmation par acteurs 207
C.5.3 Programmation par événements 207
C.5.4 Un langage dédié pour décrire un automate fini 209
C.6 Moniteur d'événements 212
C.7 Implémentation 213
C.7.1 Graphe d'événements 213
C.7.2 Événements synchrones et asynchrones 213
C.7.3 Disjonction 213
C.7.4 Langage dédié 214
C.7.5 Moniteur d'événements 215
C.8 Validation 216
C.8.1 Évaluation statique 216
C.8.2 Évaluation dynamique 216
C.8.3 Les automates finis 216
C.8.4 Moniteurs d'événements 217
C.9 Les travaux futurs 218

C.1 Introduction

Dans cette thèse, nous mettons l'accent sur la coordination de la concurrence entre les parties d'une application programmée par objets.

C.1.1 Concurrence

Dans un logiciel, la concurrence signifie que plusieurs tâches sont exécutées simultanément. Le niveau le plus fin de granularité dans la concurrence se situe au niveau fils d'exécution, connu sous le terme anglais de *thread*. Chaque thread est constitué d'une séquence d'instructions qui sont exécutées séquentiellement et qui peuvent également interagir avec d'autres fils d'exécution. Alors que la concurrence existe également entre applications, on parle alors de processus, cette thèse n'aborde que la concurrence intra-application mise en œuvre par les fils d'exécution. Sur les ordinateurs disposant de plusieurs cœurs, qui font partie qu'un seul processeur, il est possible de dédier un thread à un cœur précis et d'exécuter des *threads* simultanément sur plusieurs cœurs. Dans ce cas, les threads s'exécutent physiquement en même temps. Cependant, habituellement le nombre de threads utilisés dépasse le nombre de cœurs disponibles, dans ce cas, un cœur doit exécuter plusieurs threads simultanément. Il existe plusieurs stratégies différentes pour décider du moment où il faut changer de thread. Par exemple, un thread peut donner

le contrôle au système d'exécution, ou le système d'exécution peut reprendre le contrôle après un court laps de temps. Dans ce travail, nous dépendons de la dernière stratégie. L'ordonnancement (angl. : *scheduling*) des threads est une situation courante, parce que les ordinateurs courants avec moins de dix cœurs exécutent souvent plus d'une centaine de programmes simultanément, alors que chaque programme a besoin d'au moins un thread. Changer de thread nécessite un changement de contexte (angl. : *context switch*) qui doit stocker l'état du thread courant et charger l'état du prochain thread à exécuter. Le temps qu'un cœur passe sur les changements de contexte [93] ne peut pas être utilisé pour exécuter des programmes pour l'utilisateur.

Lorsque plusieurs threads coopèrent pour exécuter une tâche, il est nécessaire d'éviter certaines interactions pour empêcher des situations de compétition problématique [86], également connue sous le terme anglais de *race condition*. Un exemple de compétition problématique est la modification simultanée par deux threads d'une collection de nombres mutables avec l'un des threads qui double les nombres alors que l'autre les incrémente.

Le résultat de ces opérations dépend de la manière dont l'exécution des threads est entrelacée. Les valeurs obtenues sont imprévisibles, à moins qu'une synchronisation assure que les deux opérations ont lieu dans le même ordre pour tous les éléments.

Malgré la perte de performance dans le cas d'une exécution sur un seul cœur et la complexité de la conception due aux compétitions problématiques la popularité de la concurrence s'est accrue. Au début, le but de la concurrence était d'optimiser l'utilisation des ordinateurs coûteux. Désormais, celle-ci sert plutôt à fournir une meilleure expérience aux utilisateurs, comme nous l'expliquons ci-dessous. Cependant, après de nombreuses années d'études et d'utilisation la concurrence, la conception de logiciel concurrent reste une tâche complexe.

La concurrence est devenue un élément intrinsèque d'une vaste gamme d'applications logicielles. Par exemple, l'utilisateur s'attend à ce que les applications graphiques interactives restent contrôlables lorsque celui-ci tente d'exécuter une opération qui ne se termine pas instantanément. Une autre raison de faire appel à de la concurrence intra-application est que le matériel multi-cœurs actuel permet d'accélérer une programme en divisant le travail en de multiples tâches simultanées. Toutefois, le programmeur doit définir l'interaction entre les tâches qui doivent être exécutées simultanément. Contrairement à l'interaction entre les programmes distincts, l'interaction entre les parties d'une application mono-processus contourne souvent le système d'exécution, ce qui évite les surcoûts. Cependant, le système d'exécution ne peut pas contrôler la façon dont les parties individuelles interagissent. Lorsque la concurrence fait partie d'un programme, les complexités associées ne sont plus le problème exclusif des programmeurs du système d'exécution. Lorsque les interactions s'effectuent entre des cœurs répartis sur plusieurs ordinateurs, on parle d'un système *réparti* ou *distribué*. En plus de la concurrence, il faut tenir compte de la fiabilité de la communication entre les systèmes et de la réorga-

nisation parfois dynamique des machines participantes. Actuellement, l'informatique en nuage (angl. : *cloud computing*) reçoit beaucoup d'intérêt. L'interaction se situe entre des ordinateurs qui se trouvent dans différents centres de données. Les utilisateurs et les développeurs d'applications d'informatique en nuage ne sont pas influencés par des changements dans un groupe des cœurs qui interagissent et gèrent leurs données de stockage. Ceci permet d'optimiser dynamiquement les propriétés comme le prix et la vitesse. Le fournisseur d'applications peut ainsi offrir des rabais aux utilisateurs qui sont disposés à donner la priorité aux programmes d'autres utilisateurs. Si les utilisateurs sont disposés à payer davantage pour avoir leurs données plus proche d'eux ou plus près d'un endroit où celles-ci sont souvent accessibles, il est possible d'optimiser le processus de communication.

C.1.2 Supporter la concurrence dans un langage de programmation

La synchronisation peut être mise en œuvre avec des primitives de bas niveau comme les *sémaphores* [29], qui sont des compteurs explicitement gérés servant à la surveillance des opérations actives concernant les ressources partagées.
Les *moniteurs* [52, 58], situés à un niveau légèrement plus élevé, limitent le nombre de threads actifs dans une région de code, par exemple un objet. Cependant, les moniteurs restent des constructions de bas niveau qui nécessitent que les programmeurs se concentrent sur les détails, comme l'attente, pour éviter les interblocages (angl. : *deadlocks*). Afin d'utiliser correctement les primitives de synchronisation, le programmeur doit reconnaître et isoler chaque groupe d'instructions qui doit accéder aux ressources partagées et ceci de façon atomique. Ce processus complexe induit généralement les programmeurs en erreur. Ces erreurs sont souvent difficiles à comprendre et donc à mitiger.
L'histoire des langages intégrant la conception d'applications concurrentes a débuté avec des langages comme Concurrent Pascal [52], qui introduit les moniteurs comme une partie intégrale du langage. La recherche sur la concurrence a été poursuivie pendant environ 40 ans. Bien que de nombreuses solutions aient été proposées, il n'y a pas de solution générale et facile pour concevoir des applications supportant la concurrence. Par conséquent, la mise en œuvre d'interactions correctes entre des tâches concurrentes reste une tâche difficile. Le développement de nouveaux logiciels nécessite bien souvent de faire appel à la concurrence, quelle qu'en soit la difficulté.
Puisque la mise en œuvre correcte de la concurrence d'exécution avec des primitives de bas niveau est complexe, des chercheurs ont proposé des constructions linguistiques qui introduisent un niveau d'abstraction supérieur et permettent un raisonnement de plus haut niveau.
Parmi les exemples de ces concepts, nous trouvons les *join patterns* [39], les acteurs [56],

la mémoire transactionnelle [54], les futurs [7] et la synchronisation centrée sur les données [126].
Dans cette thèse, nous explorons l'utilisation des *join patterns* en particulier, parce que les autres concepts s'avèrent inadéquats dans certaines situations spécifiques.
Par exemple, la mémoire transactionnelle, qui a pour but d'isoler le programmeur de la concurrence, a un impact sur la vitesse d'exécution d'un programme. Le langage d'acteurs Erlang [6] est utilisé avec succès dans l'industrie des télécommunications. Cependant, les langages fondés sur le modèle d'acteur ne sont pas largement utilisés dans d'autres domaines. Les futurs sont principalement utilisés pour des calculs concurrents qui interagissent seulement avec les threads qui les ont lancés via leur résultat. La synchronisation centrée sur les données est une autre approche récente visant la programmation concurrente. Comme celle-ci repose sur l'identification d'unités de travail qui accèdent les données, de nouveaux programmes peuvent être structurés d'emblée de façon à bénéficier de cette approche.
Le join-calcul [39] a introduit les *join patterns* et leurs combinaisons. Nous appelons ces combinaisons des disjonctions. Elles servent à exprimer l'interaction entre un ensemble de processus qui communiquent en émettant des données via des canaux de communication.
La communication dans le modèle original est asynchrone. Toutefois, la communication synchrone peut être construite sur sa fondation asynchrone dans l'attente d'une réponse ou d'une confirmation de réception via un canal de communication.
Après son introduction, le join-calcul et les *join patterns* ont gagné beaucoup d'attention parce qu'ils combinent l'abstraction et la pratique. Le langage est assez abstrait pour surmonter les limitations de bas niveau, mais reste praticable dans une grande variété de scénarios. Pour un programmeur, les *join patterns* sont une construction unique servant à synchroniser et organiser la communication entre les parties concurrentes de logiciels. Les *join patterns* peuvent exprimer les constructions de synchronisation de bas niveau, comme les verrouillages et les sémaphores, alors qu'ils peuvent également servir à exprimer des modèles de niveau supérieur, comme des acteurs.
En raison des avantages liés aux *join patterns*, plusieurs langages ont été proposés afin de soutenir ces derniers.
Dans les prochains chapitres de cette thèse, nous référons à ces langages comme des langages de *joins* (angl. : *join languages*). Ce groupe comprend les langages JoCaml [23], Funnel [90], Join Java [60], Polyphonic C# [9], le *Join concurrency Library* [108], Scala Joins [51], JCThorn [94] et JErlang [97]. Le langage Funnel est le seul langage dans ce groupe qui ne constitue pas une extension d'un langage existant.
Dans cette thèse, nous soutenons le fait que les langages courants peuvent exprimer toute la logique des schémas de coordination mais que cette logique reste entrelacée avec la logique métier. Cette limitation est le résultat de la façon dont les canaux de communication et les émissions sont modélisées par les langages de *joins* existants.

Ces langages peuvent modéliser les canaux de trois façons, soit par un appel de méthode, soit par un appel de fonction [23, 79, 90, 9, 60], soit en utilisant explicitement les événements déclenchés [51].
Indépendamment de la représentation utilisée pour un canal de communication, l'interaction entre l'exécution du code et un ou plusieurs *join patterns* est uniquement possible via l'insertion explicit edes émissions à des points précis du code de l'application. Ces points d'émission émettent des données via un canal de communication pour une ou plusieurs disjonctions. Lorsque les méthodes ou fonctions modélisent un canal, le récepteur du canal est également codé en dur par le point d'émission.
Lorsque le code évolue après l'intégration des points d'émission, ces derniers doivent être adaptés afin d'assurer la continuité de leur fonctionnalité avec le code ajouté.
Dans le cas où les récepteurs sont définis comme une partie intégrale des points d'émission d'un canal de communication, des changements à la réception entraîneront également des changements aux points d'émissions.
La logique de coordination dans les programmes basés sur les *join patterns* est le résultat de la connexion des *join patterns* et des points d'émissions par l'intermédiaire des canaux de communication
Puisque les points d'émission sont intimement liés au restant du code, la logique de coordination est fragmentée également, ce qui crée des dépendances implicites entre les émissions, et indirectement entre les *join patterns*. Ces fortes dépendances empêchent une conception modulaire et résultent en un programme monolithique, tandis que les objectifs d'une conception modulaire ont pour but d'aider les programmeurs à comprendre et à maintenir le code et de faciliter sa réutilisation. Pour extraire la logique d'un schéma de coordination dans les programmes existants, le programmeur se doit de suivre le flux de l'application et de découvrir comment les émissions de données différentes sont reliées et interagissent entre elles. Alors que les émissions par les canaux de communication servant à exprimer la coordination entre les parties concurrentes sont mises en place par les *join patterns*, les langages de *joins* existants ne permettent que des émissions explicites qui sont mélangées avec le code implémentant la logique métier. Cela entrave la compréhension du programme et rend les programmes plus difficiles à maintenir. Cela réduit également la possibilité de réutilisation de certaines parties d'un programme.
Puisque la logique de coordination ne peut être exprimée dans un module séparé, les modifications des schémas de coordination ne peuvent être conduits sous le contrôle d'un simple raisonnement local. Les émissions observées peuvent avoir lieu dans d'autres modules.
Pour parvenir à une conception modulaire, il est nécessaire de séparer les émissions et le code de la logique métier. Ceci est similaire à l'objectif de la programmation par aspects [68] (PPA) qui tente de répartir les préoccupations transversales en modules.

C.1.3 La conception du langage JEScala

Le langage que nous proposons dans ce travail résout le problème de la coordination des parties concurrentes dans un programme à l'aide de techniques avancées de programmation. Nous exploitons la synergie entre les concepts du join-calcul et le système d'événements de pointe du langage EScala [44]. Le système original d'événements du langage EScala prend en charge deux groupes d'événements, les événements *primitifs* et les événements *déclaratifs*. Il existe deux sortes d'événements *primitifs*. La première, les événements *déclaratifs*, correspond aux événements explicitement déclenchés par des instructions dans le code. La deuxième, les événements *implicites*, correspond aux événements déclenchés lorsque l'exécution atteint un certain point dans le code.
Les événements implicites sont des points adressables dans le code, qui sont semblables aux *points de jonction* de la PPA [68]. Les événements déclaratifs utilisent des *expressions d'événements* afin de composer des événements. Parmi les exemples des opérateurs pris en charge, nous trouvons la combinaison des événements d'exécution et la transformation ou le filtrage de ceux-ci. Les expressions d'événements peuvent contenir des constructions linguistiques comme `before(m)`, qui désigne l'événement implicite qui est déclenché avant d'exécuter la méthode `m`.
Ces expressions d'événements sont une alternative aux langages de points de coupe dans la PPA. Un programmeur peut ainsi sélectionner une collection de multiples événements implicites à l'aide d'événements déclaratifs, de même qu'un *point de coupe* de la PPA permet de sélectionner des *points de jonctions*, qui correspondent à des occurrences d'événements implicites.
Les gestionnaires d'événements (angl. *event handlers*) sont des parties de code arbitraires qui sont enregistrés avec un événement et qui sont implicitement invoqués quand l'événement est déclenché. Puisque les événements EScala sont synchrones, le code qui déclenche un événement attend jusqu'à ce que tous les gestionnaires directs et indirects retournent un résultat. Les gestionnaires indirects d'un événement sont des gestionnaires qui sont enregistrés par un autre événement déclaratif et peuvent dépendre de l'événement initial les événements originaux de façon transitive au travers de leur expression d'événements.
Nous avons étendu le système d'événements d'EScala pour supporter les événements *asynchrones* de JEScala, qui ne bloquent pas l'expéditeur. Les langages de *joins*, que nous aborderons par la suite, offrent deux techniques pour envoyer des données via des canaux de communication. La première est *asynchrone*, la deuxième est *synchrone*. Les deux utilisent la même sémantique que celle utilisée dans les langages de *joins*. Les canaux de communication dans les langages de *joins* existants sont définis par la disjonction réceptrice, c'est-à-dire que le récepteur définit la synchronicité du canal. Toutefois, dans JEScala, les événements représentent les canaux de communication des langages de *joins*. Puisque ces événements supportent l'enregistrement dynamique des récepteurs, les récepteurs ne peuvent pas définir leur synchronicité. En conséquence,

la synchronicité d'un événement fait partie de la définition d'un événement primitif. Les événements déclaratifs respectent la synchronicité de l'exécution des événements qu'ils combinent. La synchronicité variable des événements déclaratifs est différente de celle des autres langages de *joins* où les disjonctions définissent la synchronicité des canaux de communication par lesquelles elles reçoivent leurs données. En utilisant par défaut l'option synchrone pour les événements primitifs, JEScala reste compatible avec le langage EScala, qui prend en charge uniquement les événements synchrones.
La puissance du système d'événements de JEScala permet de définir de façon modulaire les sources de données d'émission en même temps que la logique de synchronisation. Ceci permet de capturer des schémas de coordination complexes, qui autrement seraient éparpillés dans le code.
Ceci bénéficie non seulement à la compréhension et la maintenabilité du programme, mais permet également la réutilisation de l'implémentation d'un seul schéma de coordination avec différentes applications et vice-versa.
Puisque les schémas de coordination sont exprimés explicitement, le compilateur peut remplacer la mise en œuvre générale par une version dédiée.
Nous avons exploré cette optimisation en mettant en œuvre un langage dédié (angl. : *DSL*) qui décrit un automate fini. En plus de la mise en œuvre basée sur des événements et des *joins* à usage général, nous avons créé des versions basées sur une implémentation spécialisée pour accélérer le processus d'exécution.

C.1.4 Résumé de la contribution

En bref, cette thèse fait les contributions suivantes :

— Nous motivons la nécessité d'abstractions pour surmonter les limitations liées aux langages de *joins* (angl. : *Join languages*) actuels, qui résultent d'une logique de coordination dispersée, causée par la nécessité de définir explicitement les sources d'émission des *join patterns*.

— Nous présentons la conception de base de JEScala, un langage qui exploite une intégration transparente d'un système avancé d'événements avec des *join patterns* afin de capturer des schémas de coordination de façon expressive et modulaire.

— Nous étudions la mise en œuvre d'un automate fini (angl. : *finite state machine*). Cela aboutit à une déclaration explicite de cet automate en utilisant un langage dédié, ce qui rend possible l'utilisation d'implémentations alternatives optimisées. Nous montrons aussi des automates étendus qui mettent en œuvre un exemple de la programmation fonctionnelle réactive et un verrouillage de lecture-écriture qui prend en charge un certain nombre de lecteurs sans nécessiter une limite supérieure.

— Nous introduisons ce que nous appelons un moniteur d'événements qui restreint à une 'exécution séquentielle un ensemble de gestionnaires. L'objectif est similaire

à l'utilisation de moniteurs d'objets pour la manipulation de méthodes mutuellement exclusives dans des langages comme le Java [46], sauf que les gestionnaires ne bloquent pas l'expéditeur d'un événement.

— Nous évaluons la conception du langage JEScala, notre langage dédié pour décrire un automate fini et l'utilisation d'un moniteur d'événements. Nous commençons par la validation de notre approche en utilisant des études de cas qui montrent la validité de notre conception et nous fournissons une première évaluation de la performance de notre mise en œuvre. Ensuite, nous validons le langage dédié que nous avons conçu pour décrire des automates finis de deux façons. Premièrement, nous montrons une amélioration des métriques de code (moins de code, moins de déclenchements d'événements explicites, réduction des gestionnaires). Deuxièmement, nous utilisons une référence pour comparer différentes implémentations basées sur ce langage dédié. Enfin, nous présentons une expérience qui démontre qu'en combinant des parties d'un programme séquentiel EScala dans un programme concurrent JEScala on fait fonctionner ce dernier plus vite que son ancêtre séquentiel.

C.2 L'état de l'art

Dans le chapitre 2, nous décrivons l'état de l'art, en commençant par la programmation par aspects et la programmation par événements, suivis par un aperçu sur la programmation concurrente. Ensuite nous nous concentrons sur le join-calcul.

C.2.1 Programmation par aspects

Le but de la programmation par aspects (PPA) est de démêler la mise en œuvre de préoccupations dans le code. La programmation par objets ne peut mettre en œuvre des préoccupations transverses comme l'enregistrement d'événements dans un journal par une classe ou un autre module indépendant sans intervenir sur les classes dont on veut enregistrer les événements.

Le tissage est une métaphore populaire pour décrire la PPA. Ainsi, vous pouvez comparer les préoccupations à des fils qui font partie du tissu représentant un programme. Les langages de PPA symétriques comme HyperJ [92] sont similaires à un métier, car ils vous permettent de tisser des fils (préoccupations) de façon égale dans un programme. Cependant, les approches les plus populaires de la PPA tissent les fils (préoccupations) dans un morceau de tissu simple (constituant le programme de base), pour ensuite intégrer ces derniers dans un tissu plus complexe comme la broderie.

AspectJ [67] est le standard de fait pour la PPA. Le langage a introduit les concepts des *points de jonction*, les *points de coupe*, les *greffons* et les *aspects* qui sont également utilisés par d'autres langages PPA. Les *points de jonction* d'un programme sont des

points identifiables dans l’exécution du programme. Dans la métaphore du tissage, ce sont les points de jonction sur un morceau de tissu. Un programmeur utilise un langage dédié de *points de coupe*, où un point de coupe sélectionne un ensemble de points de coupe. Un *greffon* est un extrait de code à insérer par rapport à un ou plusieurs *points de coupe*. Il peut être inséré avant, après ou autour de chaque point de coupe sélectionné. Un aspect est une encapsulation qui associe points de coupe et greffons.

C.2.2 Programmation par événements

Dans les langages de programmation, les *événements* sont des changements observables de l’état d’un programme. Lorsqu’un événement survient, chaque observateur d’événement lance un gestionnaire qui est enregistré avec de l’événement. Ceci permet à un événement de contrôler le flux d’instructions dans un programme par événements.
L’utilisation d’événements permet de créer un code modulaire car le code qui définit un événement ne fait aucune hypothèse sur le nombre d’observateurs ni sur leur présence. Cela permet d’ajouter des gestionnaires d’événements qui font partie d’un code qui a été conçu sans aucune connaissance sur le nombre de gestionnaires à l’exécution.
Le concept sous-jacent des événements est l’*invocation implicite*. En d’autres mots, les événements invoquent leurs gestionnaires associés de façon implicite. Ceci est différent de l’appel explicite d’une méthode ou d’une fonction connue. L’usage de l’invocation implicite pour une conception modulaire a déjà été étudié auparavant par Garlan et Notkin [43].
Le patron *observateur* est une technique courante de programmation par objets pour mettre en œuvre l’invocation implicite. Toutefois, ce concept nécessite du code réutilisable qui devient de plus en plus complexe, par exemple lorsqu’un objet contient plusieurs sujets observables. Certains langages contiennent des constructions explicites pour faciliter l’écriture et la maintenance d’un tel code.
Certains cadres de programmation comme la bibliothèque de classes Java et les implémentations du MVC (modèle-vue-contrôleur) s’appuient sur le patron observateur. Lors de l’utilisation de ces cadres, l’ajout d’un *écouteur apte* à recevoir le même message provenant de plusieurs sources nécessite de la programmation supplémentaire.
Afin de réduire la complexité pour le programmeur, les langages tels que C# intègrent l’invocation implicite au langage. Le cadre de programmation portable QT, destiné à la programmation d’interfaces graphiques, intègre les événements en C++ en transformant le code C++ avec les constructions linguistiques supplémentaires `signal`, `slot` et `connect` en C++ normal.
Dans des langages comme C# et Qt, les gestionnaires sont exécutés parce que des événements sont déclenchés par le programme. Dans le traitement des événements complexes [77] (TEC, angl. : *CEP*), les événements peuvent être déclenchés par d’autres événements. Ceci permet la détection d’événements de haut niveau basés sur des événements de bas niveau. Par exemple, le son d’une porte qui se ferme suivi par les pas d’une

personne, signifie que quelqu'un est entré. Rapide [120] est le premier système de TEC avec son propre langage. SASE [129], Cayuga [27] et Esper [35] sont accessibles par d'autres programmes à l'aide de requêtes. Les requêtes sont semblables à des requêtes SQL, mais au lieu d'extraire les données de tableaux, ils les extraient d'un flux d'événements. Le langage EventJava [37] est une extension du langage Java qui traite des flux d'événements. Les systèmes de TEC sont capables de transformer et de filtrer des événements, de détecter les séquences et ils supportent également des fenêtres coulissantes sur un flux d'événements.

C.2.3 La combinaison d'événements et d'aspects

La programmation par aspects et la programmation par événements comportent des éléments similaires qui peuvent mutuellement se renforcer. Les deux paradigmes utilisent une terminologie différente, mais Steimann et al. [114] ont décrit la relation entre les concepts des deux paradigmes, par exemple la similarité entre un *événement* et un *point de jonction*. Ce rapport est utilisé par les langages de programmation comme Ptolemy [103], JasCo [116] et EventCJ [65] pour mettre en œuvre des points de jonction au moyen d'événements et de greffons par des gestionnaires de tels événements.

C.2.4 EScala

Le langage EScala intègre un système avancé d'événements dans Scala, un langage de programmation par objets qui permet aussi la programmation fonctionelle. EScala prend en charge la PPA en utilisant un système d'événements avancé et dédié. Dans le langage EScala, les événements sont des membres des objets. Dans ce cas, le mot événement fait référence à la construction syntaxique qui peut être déclenchée pour émettre une *occurrence d'événement*. Nous utilisons souvent le terme événement pour désigner une occurrence d'événement sauf s'il y a risque d'ambiguïté.
Le système d'événements prend en charge différentes sortes d'événements, à savoir les événements *primitifs* et les événements *composés*. Il existe deux sortes d'événements *primitifs* : les événements *impératifs* qui sont explicitement déclenchés par le programme et les événements *implicites* qui sont déclenchés lorsqu'un certain moment précis est atteint dans l'exécution du programme. Les événements implicites ne sont pas définis par les programmeurs, mais ils sont une partie intégrale du programme.
Les événements *composés* sont déclenchés en raison d'un autre événement. Ils sont définis par des *expressions d'événements déclaratifs*. Nous mentionnons trois opérateurs qui sont utilisés pour construire des expressions d'événements. Pour transformer un événement, nous utilisons l'opérateur `map` qui applique une fonction sur chaque occurrence de l'événement. L'opérateur `&&` filtre les résultats si l'occurrence satisfait un prédicat. Le troisième opérateur est l'union (`||`) pour combiner les occurrences des événements de ses opérandes.

L’exécution d’une méthode `m` peut être observée par les expressions `before(m)` et `after(m)`. Les événements résultants exposent les arguments de la méthode. En plus, l’événement `after` fournit aussi la valeur retournée par la méthode exécutée.

C.2.5 La concurrence

Lorsque plusieurs threads collaborent pour effectuer une tâche, les threads doivent être coordonnés afin d’empêcher des compétitions problématiques [86]. Le résultat d’une compétition problématique dépend de la façon dont l’exécution des threads est entrelacée.
Dans les sections suivantes, nous donnons un aperçu des différents modèles de programmation qui permettent de concevoir des programmes concurrents en contrôlant les entrelacements.

Synchronisation par mémoire partagée Dans le modèle de mémoire partagée, plusieurs tâches partagent leur mémoire, ou une partie de celle-ci, généralement le tas. Les processeurs récents à plusieurs cœurs peuvent exécuter des tâches en parallèle, y compris accéder à la mémoire partagée. Un programme séquentiel peut permuter les valeurs de deux variables en utilisant une variable auxiliaire. Lorsque deux tâches utilisent la même technique sur les mêmes variables, on s’attend à ce que les variables soient restaurées à leurs valeurs initiales. Lorsque ces tâches s’exécutent simultanément, le résultat dépend de l’entrelacement des instructions exécutées par les deux tâches. Ce type de compétitions problématiques est évité par la synchronisation des accès aux données dans la mémoire partagée.
La synchronisation peut être mise en œuvre avec des primitives de bas niveau, comme les *sémaphores* [29], qui sont des compteurs explicitement gérés servant à surveiller les opérations actives utilisant les ressources partagées. À un niveau légèrement plus élevé se situent les *moniteurs* [52, 58]. Cependant, les moniteurs sont encore des constructions de bas niveau qui nécessitent que les programmeurs se concentrent sur les détails, comme l’attente de notifications, pour éviter les interblocages. L’utilisation correcte des primitives de synchronisation est complexe et induit facilement des erreurs avec un code qui est difficile à comprendre et donc à maintenir.

Synchronisation par envoi de messages Le modèle sans partage, ne repose pas sur des primitives de bas niveau, mais évite de partager les données mutables en communiquant par envois de messages plutôt qu’au travers de la mémoire partagée. Ci-dessous nous décrivons le modèle d’acteur [56] et les futurs [7], qui sont des exemples du modèle sans partage.

Modèle de l'acteur Dans ce modèle [56, 2], une tâche qui est effectuée par un *acteur*, qui dispose de sa propre mémoire locale et communique en envoyant des *messages asynchrones* à d'autres acteurs. Il n'y a pas de concurrence dans un acteur. Mais deux ou plusieurs acteurs peuvent s'exécuter simultanément. Chaque acteur a une boîte aux lettres unique dans laquelle il lit les messages qui lui sont destinés avant d'y réagir.

Les futurs Les futurs fournissent une forme de concurrence implicite aux programmeurs. Les futurs dans les langages actuels comme Scala [91] correspondent à des calculs encapsulés qui sont exécutés par un autre thread que celui qui les définit. La définition d'un futur renvoie le résultat du calcul dans un paramètre fictif (angl. : *placeholder*). Une fois que le résultat est disponible, ce paramètre fictif donne accès à ce résultat. Lorsque le programme a besoin d'un résultat qui n'est pas encore disponible, le programme attend jusqu'à ce que celui-ci devienne disponible.

La mémoire transactionnelle Comme dans les bases de données, les threads accèdent à la mémoire, de manière optimiste, au sein de transactions. Si un accès concurrent sur les mêmes données est détecté, la transaction échoue et l'état initial est restauré. Des implémentations efficaces de la mémoire transactionnelle [54] (angl. : *Transactional Memory*) sont difficiles.

C.2.5.1 Joins

Le join-calcul [39] (angl. : *Join Calculus*) introduit les *join patterns* et leurs combinaisons, que nous appelons des *disjonctions*, dont le concept-clé est d'exprimer l'interaction entre un ensemble de processus qui communiquent en émettant des données via des canaux de communication. La communication dans le modèle original est asynchrone. Toutefois, la communication synchrone peut être fondée sur son équivalent asynchrone dans l'attente d'une réponse ou d'une confirmation d'envoi des données.
Les *join patterns* observent les données disponibles via un ou plusieurs canaux. Lorsque des données sont disponibles via tous les canaux d'un *join pattern*, le *join pattern* lit celles-ci par une action atomique et les mets à la disposition de la réaction engendrée par la reconnaissance du *join pattern*. La réaction est un processus. La définition de processus est suffisamment souple pour démarrer zéro ou plusieurs processus. Dans le reste de cette section, nous utilisons le mot *événement* lorsque les données sont disponibles par l'intermédiaire d'un canal.
Les *join patterns* dans une disjonction observent les canaux simultanément. Par conséquent, les modes de test et de lecture des données de tous leurs canaux s'effectuent dans une opération atomique. Lorsque dans une disjonction, deux ou plusieurs *join patterns* observent le même canal, des situations de compétition peuvent survenir entre les *join patterns* qui ont accès aux événements du canal partagé.

Langages de joins Les *join patterns* et les événements asynchrones sont des éléments qui font partie de langages de programmation que nous appelons langages de *joins* (angl. : *Join languages*). Ces langages représentent les canaux de join-calcul de manières différentes. Certains langages de *joins* utilisent le mot *événement* (angl. : *event*) pour désigner les données envoyées par un canal ou pour le canal lui-même. Toutefois, aucun des langages de *joins* dont nous avons parlé, ne comporte des événements à invocation implicite.
JoCaml [23] est le premier langage de programmation conçu pour soutenir le join-calcul. Funnel [90], qui est proche du Scala, a été développé par la suite. Ce langage a été conçu sur base d'une variante du join-calcul, le join-calcul par objets (angl. : *object-based Join calculus*). Les langages Join Java [60] et Polyphonic C# [9] utilisent des objets pour définir des disjonctions. Cω[106] (C omega) est une extension de C#, qui intègre toutes les extensions de Polyphonic C# et d'autres comme l'extension linq [121] pour l'accès aux données. La *Joins library* [108] pour la plate-forme .NET est la première bibliothèque conçue pour les *join patterns*. Des optimisations comme le verrouillage à granularité fine ont été étudiées dans la mise en œuvre ultérieure de cette bibliothèque [122]. Concurrent Basic [109] intègre la bibliothèque susmentionnée dans le langage Visual Basic. En plus de cette intégration, ce langage prend en charge l'héritage. L'accent de la bibliothèque ScalaJoins est mis sur l'intégration de *join patterns* en Scala. JErlang [97] est une extension du langage d'acteurs Erlang [5] qui introduit des *join patterns*.

Le rôle des joins patterns dans la conception de programmes Les langages de *joins* intègrent deux constructions : les canaux asynchrones, qui ne bloquent pas leur expéditeur, et les *join patterns*, qui synchronisent la réception des données qu'ils reçoivent.
Les langages de *joins* permettent d'implémenter des constructions classiques de la programmation concurrente comme les verrous, les acteurs et les automates finis. Par exemple, un verrouillage peut être exprimé comme un *join pattern* entre un appel d'une méthode synchrone de requête et un appel d'une méthode asynchrone qui exprime l'état libre (angl. : *free state*) lorsque le verrouillage est terminé. Le canal asynchrone est utilisé une première fois afin d'initialiser le verrouillage. Les *join patterns* peuvent mettre en œuvre un acteur à l'aide d'un disjonction de *join patterns*. Il y a un *join pattern* par type de message reçu par l'acteur, qui combine un canal asynchrone pour le type de message traité ainsi qu'un canal synchrone, identique pour tous les *join patterns*. Un envoi en boucle sur le canal synchrone garantit qu'un unique *join pattern* est actif à tout instant et par là synchronise la réception et le traitement des messages. Pour mettre en œuvre des automates finis au moyen de *join patterns* il faut utiliser des canaux asynchrones qui demeurent en attente jusqu'à ce que l'état de l'automate change, ce que nous appelons des *canaux d'état*. Un *join pattern* peut encoder une transition en joignant un canal asynchrone d'état et un *canal d'action*. Ensuite, la méthode asynchrone de l'état cible est déclenchée. En général, il n'y a pas d'arguments pour des méthodes

```
object OB { // Online Booking App
  def handle(...)={
    CL.OB_beforeHandle()
     ..
  }
}
object MP { // Market Place App
  def handle(...)={
    CL.MP_beforeHandle()
     ..
  }
}
```

FIGURE C.1 – Applications web et les méthodes qui reçoivent les requêtes.

d'état ni pour les méthodes d'action.

C.3 La problématique

Nous nous basons sur une étude de cas pour illustrer les lacunes des langages de *joins*. Dans cette étude de cas, nous utilisons un serveur qui héberge des applications web, que nous voulons empêcher d'utiliser toutes les ressources, puisque le serveur exécute également d'autres applications que nous ne voulons pas dépourvoir de ressources.
Dans l'exemple suivant, nous utilisons deux applications. L'une est désignée comme *Online Booking* (OB) et l'autre comme *Market Place* (MP). La figure C.1 illustre les applications et la méthode pertinente.

La structure des applications Dans des cadres de programmation comme J2EE, après l'analyse d'une requête http, une méthode conçue par un programmeur est appelée. Nous appelons cette méthode `handle` et omettons les détails de ses arguments. Nous limitons le nombre d'appels de la méthode `handle` durant un intervalle afin de protéger les ressources disponibles. Les appels multiples à deux méthodes peuvent être simultanées. Ceci permet à un seul navigateur d'envoyer plusieurs requêtes simultanément ou de traiter des requêtes concurrentes de plusieurs utilisateurs.

Instrumentation Afin d'expliquer comment cela peut fonctionner avec un langage de *joins*, nous devons introduire une méthode `createToken` de génération de jetons qu'est appelée à un intervalle fixe. Chaque fois que cette méthode est appelée, un seul appel à une des méthodes `handle` a lieu. Cette horloge est mise en œuvre par l'objet

```
1 object TG extends Thread { // Token Generator
2   def createToken()={
3      CL.unblock()
4       ...
5   }
6   def override def run = while(true) {
7     sleep(1000) // sleep one second
8     createToken()
9   }
10 }
```

FIGURE C.2 – Création de jetons à intervalle fixe.

`TG` (*token generator*) illustrée dans la figure C.2. L'objet `TG` s'exécute dans son propre thread en appelant la méthode `createToken`.
La logique de coordination intervient grâce à l'instrumentation que nous avons ajoutée aux applications web (Figure C.1 Lignes 3 et 9) et au *token generator* (Figure C.2 Ligne 3).

Logique de coordination La logique de coordination est mise en œuvre par l'objet `CL` (Figure C.3). Cet objet combine les requêtes émanant des applications web par l'intermédiaire des méthodes `OB_beforeHandle` (Ligne 3) et `MP_beforeHandle` (Ligne 6). Ces deux méthodes appellent à leur tour la méthode `mayBlock`. Cette dernière méthode renvoie à la méthode `RL.block` si les ressources disponibles dépassent le maximum autorisé (Ligne 10). Lorsque le système est surchargé, nous utilisons un *limiteur de taux* (angl. : *rate limiter*) qui est mis en œuvre par l'objet `RL`, que nous expliquons ci-dessous. La logique de coordination transmet également ces appels de méthodes à sa méthode `unblock` (Ligne 13) à partir du *token generator* au limiteur de taux (Ligne 14).

Limiteur de taux La figure C.4 montre la mise en œuvre du limiteur de taux dans le langage synthétique Polyphonic Scala. Ce dernier est une extension du langage Scala similaire à l'extension Polyphonic de C#. Les *join patterns* sont implémentés comme des méthodes contenant plusieurs en-têtes (angl. : *headers*) jointes par une esperluette (`&`). Quand tous les en-têtes ont fait l'objet d'un appel, le corps est exécuté. L'objet résultant (`RL`) agit comme une disjonction implicite pour ses *join patterns*. Un seul en-tête de méthode avec les mêmes arguments et le même type de réponse peut faire partie de plusieurs *join patterns*. L'autre extension est l'utilisation du mot-clé `async` comme s'il s'agissait d'un type de réponse. L'invocation de la méthode qui renvoie le type `async` n'attend pas la fin de son l'exécution pour continuer avec l'instruction

```
1  // Coordination Logic
2  object CL {
3      def OB_beforeHandle()={
4          mayBlock()
5        }
6      def MP_beforeHandle()={
7          mayBlock()
8        }
9      def mayBlock()={
10         if (systemLoadHigh())
11           RL.block()
12     }
13     def unblock()={
14         RL.unblock()
15     }
16 }
```

FIGURE C.3 – Logique de coordination.

```
1  object RL {
2      def block():Unit & free():async {
3          busy() // free to busy
4      }
5      def unblock():async & busy():async {
6          free() // busy to free
7      }
8      def unblock():async & free():async {
9          // Stat.toFree() //future extension
10         free() // absorbed unblock
11      }
12     free() // initial state in constructor
13 }
```

FIGURE C.4 – Limiteur de taux mis en œuvre par un automate fini.

suivante.
Dans la figure C.4, l'automate fini (angl. : *FSM*) expose deux méthodes, à savoir la méthode `block` (Ligne 2) et la méthode `unblock` (Lignes 5 et 8). Nous appelons ces méthodes des méthodes d'*action*. Pour le langage de programmation il n'y aucune distinction entre les méthodes d'*action* et les autres méthodes. Les états sont créés par des méthodes asynchrones, dites *méthodes d'état*, en attente. La ligne 12 lance l'appel de méthode pour l'état initial. Les trois *join patterns* traitent les combinaisons pertinentes d'appels à une méthode d'action et une méthode d'état et relancent un nouvel appel pour chaque changement d'état. Puisque les méthodes d'action apparaissent en premier dans chaque *join pattern*, le premier *join pattern* sur la ligne 2 peut être vu comme faisant passer l'automate de l'état libre (`free`) à occupé (`busy`) suite à l'action de blocage (`block`).

Discussion Le schéma de coordination est simplement mis en œuvre par l'objet `CL` et le limiteur de taux. Toutefois, la mise en œuvre dépend de la modification des applications web. Une mise à jour d'une des applications web est possible uniquement après l'instrumentation de la nouvelle version. L'ajout d'une nouvelle application web demande aussi d'étendre la logique de coordination. Toute modification de code métier requiert de bien comprendre ses possibles interférences avec la logique de coordination sous peine de dysfonctionnement.

C.4 Un système d'événements avancé à la rescousse

La capture de certains points dans l'exécution d'un programme est une approche commune en matière de programmation par aspects (PPA), qui est basée sur la sélection de points de jonction à l'aide de points de coupe. En PPA, les points de jonction permettent d'observer le moment où un traitement de requête est sur le point d'être exécuté. Les points de coupe peuvent être utilisés pour combiner les points de jonction et pour réagir à des conditions spécifiques, telles qu'une charge élevée du système. Pour cette raison, nous croyons que l'extension à la PPA de Polyphonic Scala conduirait à une programmation plus robuste de notre étude de cas. Des travaux antérieurs sur les langages séquentiels ECaesarJ [88] et EScala [44] suggèrent d'unifier programmation par aspects et programmation par événements. Ces langages introduisent le concept du déclenchement implicite d'événements par l'exécution d'un programme comme point de jonction. Ils prennent également en charge des événements explicites déclenchés par une instruction. La troisième option est l'utilisation d'événements composés grâce à l'introduction d'opérateurs logiques dans les expressions d'événements.
Afin d'intégrer les *join patterns* avec le système d'événements décrit, nous considérons l'émission de données sur un canal comme un événement explicite et l'opérateur join comme un opérateur additionnel servant à composer les événements.

event-decl : := prim-event | *decl-event*
prim-event : := **imperative** *[sync-modifier]* **evt** *event-name*
decl-event : := **evt** *event-name* **=** *event-express*
| **evt (***event-name { , event-name}*$^+$ **)** =
(*join-express {* **|** *join-express}*$^+$ **)**
event-express : := [obj-ref **.** *] event-name*
| ***super*** **.** *event-name*
| *event-prefix-operator event-express*
| *event-express event-infix-operator event-express*
| *event-express fun-operator fun*
| *implicit-event*
implicit-event : := [obj-ref **.** *] implicit-selector* **(** *method-name* **)**
implicit-selector : := **beforeSync** | **afterSync**
| **beforeAsync** | **afterAsync**
| **before** | **after**
sync-modifier : := **sync** | **async**
event-prefix-operator : := **!!**
event-infix-operator : := **||** | **&** | ...
fun-operator : := **map** | **&&** | ...
join-express : := **(** *event-name {* **&** *event-name }*$^+$ **)**
method-modifier : := **observable**

FIGURE C.5 – Éléments spécifiques de la syntaxe de JEScala.

L'intégration de ces idées au langage EScala conduit à notre langage : JEScala. Figure C.5 montre les extensions qui doivent être apportées à la syntaxe de Scala.

C.4.1 Événements impératifs

Les déclenchements d'événements explicites font l'objet d'une construction linguistique pour mettre en œuvre l'invocation implicite, que l'on retrouve également dans d'autres langages comme C#. Ces déclarations commencent par les mots-clés `imperative evt` éventuellement suivis d'un modificateur `sync` ou `async`, puis le nom de l'événement entre crochets. Les événements impératifs sont synchrones par défaut. Ils sont semblables aux émissions par un canal synchrone dans les langages de *joins*, où l'émission d'un événement attend une réponse du receveur/gestionnaire de l'événement.

C.4.2 Événements implicites

Une alternative au déclenchement explicite d'événements dans le langage EScala, est l'utilisation du déclenchement implicite. Avec le déclenchement implicite, les événements sont déclenchés lorsque le programme atteint un point spécifié dans son exécution. Dans les langages de PPA, de tels points sont appelés des points de jonction.
Dans le langage EScala, il est possible de marquer les méthodes avec le modificateur `observable` pour exposer les événements implicites qui indiquent le début et la fin de leur exécution.
Pour les méthodes observables, le langage EScala fournit deux événements implicites `before(`*nom-de-méthode*`)` et `after(`*nom-de-méthode*`)` qui sont déclenchés quand l'exécution de la méthode commence ou se termine. JEScala expose quatre événements implicites supplémentaires pour une méthode : `beforeSync`, `beforeAsync`, `afterSync` et `afterAsync`.
Tous ces événements implicites réfèrent à une méthode par son nom. Les événements `beforeSync(`*nom-de-méthode*`)` et `afterSync(`*nom-de-méthode*`)` sont des synonymes de before `before(`*nom-de-méthode*`)` et `after(`*nom-de-méthode*`)`, ils ont la même sémantique.

C.4.3 Événements déclaratifs

Les opérateurs, dont les principaux sont l'union (`||`), la transformation (`map`) et le filtre (`&&`), servent à construire des expressions d'événements. L'union de deux événements émet les occurrences de deux opérandes, qui à leur tour sont des événements. L'opérateur map émet les occurrences du premier opérande, constituant ainsi un événement après que le second opérande, constitué d'une fonction, est appliqué sur son paramètre. Les résultats de l'opération de filtre constituent un événement qui émet les occurrences

du premier opérande lorsque le second opérande, un prédicat, appliqué au paramètre de l'occurrence de l'événement est satisfait. Les événements déclaratifs ne modifient pas la synchronicité des occurrences reçues de ses opérandes d'événement et donc une union peut faire varier la synchronicité de l'événement résultant.

C.4.4 Événements abstraits

Une classe abstraite peut déclarer des événements par leur nom et type d'argument et déléguer leur définition à une sous-classe. Ces déclarations ne sont pas en mesure de spécifier la synchronicité des événements, étant donné que ce pourrait engendrer des conflits dans la mesure où la synchronicité d'une expression d'événement n'est en général connue qu'à l'exécution.

C.4.5 Join patterns avec des événements implicites et déclaratifs

L'exemple de la figure C.1 nécessitait des instructions supplémentaires (Lignes 3 et 9) pour permettre à la logique de coordination de fonctionner. Ici, les événements implicites JEScala `OB.beforeSync(handle)` et `MP.beforeSync(handle)` peuvent être utilisés au sein de la logique de coordination sans qu'il soit nécessaire de modifier les programmes des applications web. De plus, la ligne 3 dans la figure C.2 peut être remplacée par `TG.beforeAsync(createToken)`. Ici nous utilisons l'événement asynchrone, puisque la chaîne d'appels de méthode dans la figure C.4 se termine par un en-tête de méthode asynchrone `free` (Lignes 5 et 8). En JEScala, l'émetteur décide de la synchronicité, tandis que dans d'autres langages de *joins* cette décision est prise par la disjonction de réception.
La figure C.6 illustre la mise en œuvre en JEScala du code des figures C.4 et C.3. Les lignes 1–5 définissent deux événements déclaratifs. L'événement `block` (Ligne 2) combine les deux événements synchrones implicites dans une union filtrée par le prédicat `systemLoadHigh` (Ligne 3). L'événement `unblock` (Ligne 4) est un alias pour un événement implicite asynchrone.
Les lignes 7–17 dans la figure C.6 illustrent l'implémentation en JEScala du limiteur de taux (RL). Les deux événements d'état, `free` (Ligne 8) et `busy` (Ligne 9), sont définis explicitement plutôt qu'implicitement. En Polyphonic Scala, une disjonction est définie implicitement par un objet, mais en JEScala une disjonction est une construction composée explicitement (Lignes 10 – 12) par le biais d'un opérateur `|`.
Chaque *join pattern* résulte en un événement qui fait partie d'un n-uplet, défini à la ligne 10. L'état `free` est atteint par les transitions mises en œuvre aux lignes 11 et 12, qui déclenchent les événements `freed` et `absorbed`, généralisés en l'événement `toFree` (Ligne 13). Une solution alternative serait de gérer ces deux événements avec le même gestionnaire. Les lignes 14 et 15 déclenchent l'événement asynchrone pour le nouvel état. La dernière ligne déclenche l'événement pour l'état initial (`free`).

```
1  object CL {
2    evt block = ( OB.beforeSync(handle) || MP.beforeSync(handle) ) &&
3                   systemLoadHigh()
4    evt unblock =  TG.beforeAsync(createToken)
5  }
6
7  object RL {
8    imperative async evt free[Unit]
9    imperative async evt busy[Unit]
10   evt (toBusy, freed, absorbed) = ( CL.block   & free
11                                   | CL.unblock & busy
12                                   | CL.unblock & free  )
13   evt toFree = freed || absorbed
14   toBusy += ((arg:Any)=>busy())
15   toFree += ((arg:Any)=>free())
16   free() // initial state
17 }
```

FIGURE C.6 – Implémentation de CL et RL avec JEScala.

En JEScala, une disjonction est une construction explicite ; elle n'est pas définie implicitement par un objet. Ceci permet d'utiliser plusieurs disjonctions pour exprimer les interactions complexes d'un unique objet. Cela élimine également la nécessité d'introduire des objets synthétiques pour exprimer des disjonctions.

C.4.6 L'enregistrement dynamique

Pour enregistrer le temps passé par l'automate dans l'état `busy`, nous pouvons associer des gestionnaires supplémentaires aux événements `block` (Ligne 2) et `unblock` (Ligne 4) de l'objet `CL` (Figure C.6). Ceci entraîne une charge de travail supplémentaire, puisque le temps doit être compté entre chaque déclenchement d'un événement `block` et le déclenchement de l'événement `block` suivant. Il est donc recommandé de désactiver cette fonction lorsque l'information n'est pas utile, ce qui peut être fait en JEScala grâce à la possibilité d'ajouter et retirer dynamiquement un gestionnaire d'événements de la liste des gestionnaires associés à un événement donné. Les autres langages de *joins* sont contraints d'insérer de nouveaux appels conditionnels au sein des méthodes de l'objet `CL`. La gestion de la coordination et la gestion du temps sont melangées.

C.5 Automate fini

Les automates finis sont un moyen d'expression de la coordination de haut niveau. Dans notre étude de cas, nous avons utilisé un automate fini pour mettre en œuvre la classe `RL`. Les automates finis sont définis par des transitions qui changent l'état en raison d'une action. Nous commençons par un aperçu des différentes implémentations possibles d'un automate fini. Ensuite nous montrons comment utiliser les événements de JEScala dans une implémentation basée sur les *join patterns*. Dans les programmes concurrents, les actions qui gèrent un automate fini peuvent être concurrentes. Ceci nous oblige à mettre en œuvre des automates finis sécurisés vis-à-vis des threads (angl. : *thread-safe*).

C.5.1 Programmation par objets

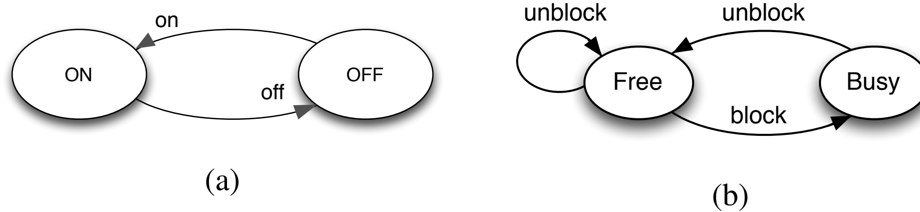
Le patron état (angl. : *state*) [42] est une manière courante de mettre en œuvre un automate fini.
Le principe du patron consiste à définir sous la forme d'une classe abstraite ou d'une interface le type des états ainsi que les actions applicables sous la forme de méthodes. Pour chaque état possible, une classe singleton définit le comportement de chacune des méthodes, en particulier la modification d'une variable d'état partagée qui enregistre l'état courant de l'automate. Dans un cadre séquentiel, les actions impossibles dans un état donné sont implémentées par des méthodes qui lèvent une exception. Dans un cadre concurrent, ces actions sont implémentées par des méthodes gardées à l'aide d'un moniteur qui garantit que les transitions s'effectuent de manière atomique. Il est également possible d'attendre une autre action lorsqu'une action, qui n'a pas abouti à une transition, a été reçue. En général, les implémentations basées sur le patron état bloquent l'appelant jusqu'à ce que l'action soit traitée.

C.5.2 Programmation par acteurs

Un automate fini peut être implémenté à l'aide d'un acteur qui reçoit pour messages les actions de l'automate. Puisque les acteurs traitent les messages qu'ils ont reçus dans leur boîte aux lettres de façon séquentielle, aucune synchronisation supplémentaire n'est nécessaire. Comme les messages, une fois placés dans la boîte aux lettres, sont traités de manière asynchrone par l'acteur, l'appelant n'attend pas, par défaut, que l'action soit traitée.

C.5.3 Programmation par événements

La figure C.7a illustre un automate fini qui fonctionne comme un interrupteur marche/arrêt (angl. : *on/off switch*). La figure C.7b reprend l'exemple du limiteur de taux, plus complexe, utilisé dans notre étude de cas.



(a) (b)

FIGURE C.7 – Automates finis de l'interrupteur on/off (a) et du limiteur de taux (b).

```
1  class Switch {
2     outer =>
3     imperative async evt on[Unit]
4     imperative async evt off[Unit]
5     private var currentState:State = OFF
6     private trait State {
7        def on: Unit
8        def off: Unit     }
9     private def await(cond: => Boolean) = while (!cond) { wait() }
10    private object ON extends State {
11       def on =  await (currentState == OFF)
12       def off = { currentState = OFF
13                   outer.notify() }
14    }
15    private object OFF extends State {..}
16    on  += ((_) => synchronized { currentState.on  } )
17    off += ((_) => synchronized { currentState.off } )
18 }
```

FIGURE C.8 – Interrupteur concurrent avec patron état, moniteur et événements.

Événements La figure C.8 illustre les événements asynchrones `on` (Ligne 3) et `off` (Ligne 4) qui remplacent les méthodes associées aux actions ou les messages d'acteur dans les implémentations précédentes.
Cette implémentation reste toutefois très proche de l'implémentation concurrente de base en programmation par objets avec l'utilisation d'un moniteur pour protéger les modifications apportées à la variable `currentState` (Ligne 5) qui contient l'état courant de l'automate fini.
Le trait `State` (Lignes 6–8) déclare deux méthodes abstraites pour gérer les actions `on` et `off`. La méthode `await` (Ligne 9) est utilisée lorsqu'un événement d'action reçu ne peut pas être traité dans l'état actuel et attend un état qui peut accepter l'événement. Nous démontrons la mise en œuvre de l'état `on` (Lignes 10–14). La prise en compte de l'action `on` (Ligne 11) est mise en attente jusqu'à ce que l'état devienne `OFF`. La gestion de l'action `off` consiste à faire passer l'état à `off` et avertir les threads en attente de la méthode await en passant par le moniteur d'objet. Nous ne montrons pas l'objet `OFF` en détail puisqu'il est analogue à l'objet `ON`. Les gestionnaires des événements `on` (Ligne 3) et `off` (Ligne 4) sont enregistrés respectivement (Lignes 16 et 17). Ils se synchronisent via le moniteur sous-jacent.
Un événement d'action dans la figure C.8 peut être remplacé par un événement synchrone. En sélectionnant la concurrence désirée des événements d'action, la coordination peut bloquer ses clients si nécessaire, par exemple pour protéger l'accès à une ressource partagée.
Le code de notre exemple est basé sur l'invocation d'événements impératifs. Un autre choix est d'utiliser des événements implicites permettant de coordonner du code qui ne doit pas être modifié par l'insertion explicite d'événements. Finalement, il est aussi possible de passer les événements `on` et `off` en paramètre du constructeur de la classe `Switch`. La synchronicité des événements dépendra alors des arguments utilisés à l'instanciation.

Join patterns et événements Nous renommons les actions et les états de la figure C.7a et nous ajoutons une boucle commençant par et retournant l'état `Free` pour traiter les événements `unblock`. La figure C.7b montre une implémentation directe de l'automate fini de la figure C.7b, indépendamment de l'existence d'un objet `CL`. Alors que dans notre étude de cas (voir Figure C.6), les événements `block` et `unblock` étaient définis dans un objet externe. Ils sont ici définis par l'objet implémentant l'automate. Le reste de l'implémentation est similaire et a déjà été décrite précédemment.

C.5.4 Un langage dédié pour décrire un automate fini

La mise en œuvre d'un automate fini (par exemple dans la figure C.9) contient plus de code que la description des états et des transitions. Modifier les transitions dans une

```
object FsmRL {
  imperative sync evt  block[Unit]
  imperative async evt unblock[Unit]
  imperative async evt free[Unit]
  imperative async evt busy[Unit]
  evt (toBusy, freed, absorbed) = ( block   & free
                                  | unblock & busy
                                  | unblock & free )
  evt toFree = freed || absorbed
  toBusy += ((arg:Any)=>busy())
  toFree += ((arg:Any)=>free())
  free() // initial state
}
```

FIGURE C.9 – Répetition d'automate finis RL avec JEScala.

telle application peut introduire des incohérences. Par conséquent, nous avons conçu un langage dédié pour décrire explicitement les automates finis et éviter le code répétitif.

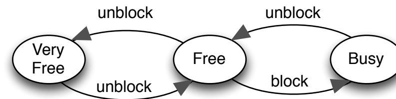
Définition d'un automate fini La figure C.10 illustre la définition d'un automate fini à l'aide notre langage dédié au travers de l'exemple du limiteur de taux. L'exemple illustre la possibilité d'associer l'exécution de code à une transition.
Dans le langage dédié, les états sont mis en œuvre comme des instances d'un supertype commun. Dans la figure C.10, ce supertype est défini par le trait `S` (Ligne 1). Les états `Free` (Ligne 2) et `Busy` (Ligne 3) sont des objets singleton qui héritent du trait `S`.
Les transitions de l'automate (Lignes 4–10) ainsi que son initialisation (Ligne 9) sont décrits dans le constructeur primaire de la classe `RateLimiter`. Le trait `FSM_J`, paramétré par le type des états, définit les méthodes nécessaires à la création des transitions à base de *join patterns*. Son argument type spécifie le type commun `S` pour les états.
L'événement impératif `waiting` (Ligne 5) est déclenché par la seconde transition (Ligne 7). Les lignes 6–8 décrivent trois transitions constituées d'un état source, d'un état destination et d'une action séparés par une flèche et le mot-clef `on`. Dans l'exemple nous utilisons les événements d'un client `cl` comme événements d'action.
Le client `cl` est un argument du constructeur primaire de la classe `RateLimiter` (Ligne 4). La première transition (Ligne 6) exécute une fonction sans arguments. Lors de l'exécution de la deuxième transition (Ligne 7), l'événement `wait`, qui est défini sans argument à la ligne 5, est déclenché. La dernière ligne du constructeur indique l'état initial (`Free`). Le programmeur indique implicitement que la description est complète en appelant la méthode `initialState`. Ceci « gèle » la représentation interne des transitions et génère les structures qui mettent en œuvre l'automate fini.

```
1 trait S
2 object Free extends S
3 object Busy extends S
4 class RateLimiter(val cl:Client) extends FSM_J[S] {
5   imperative async evt waiting[Unit]
6   Busy -> Free on cl.unblock apply(()=>println("freed") )
7   Free -> Busy on cl.block triggers waiting
8   Free -> Free on cl.unblock
9   initialState(Free)
10 }
```

FIGURE C.10 – L'automate fini avec notre langage dédié.

```
1 object VeryFree extends S
2 class Modified(val cl:Client)
3          extends RateLimiter(cl) {
4   unfreeze
5   remove Free -> Free on cl.unblock
6   Free -> VeryFree on cl.unblock
7   VeryFree -> Free on cl.unblock
8   initialState(Free)
9 }
```

(a)



(b)

FIGURE C.11 – Ajouter l'état VeryFree au automate RL (a) et le résulte (b).

Modifier des automates finis avec l'héritage Notre langage dédié permet de modifier un automate par héritage. À titre d'exemple, la figure C.11b étend le limiteur de taux de la figure C.7b avec l'état `VeryFree`.
La figure C.11a montre comment l'héritage étend la description de la figure C.10. D'abord, le constructeur de la sous-classe appelle la méthode `unfreeze` (Ligne 4) pour autoriser des modifications des transitions collectées et pour désactiver les structures générées par le constructeur de la superclasse. En ajoutant une transition avec le mot-clé `remove` (Ligne 5), il est possible de supprimer une transition décrite par la superclasse. La description d'une transition (Ligne 6) est ajoutée à l'automate fini résultant. Enfin, le nouvel automate fini est construit après la sélection de l'état initial (Ligne 8).

La conception du langage dédié Le langage dédié comprend deux couches. Le trait `FSM` constitue la première couche qui recueille les transitions qui sont utilisées par la deuxième couche. Le trait `FSM_J` étend le trait `FSM` qui crée les structures de données

pour mettre en œuvre l'automate fini, constituant ainsi la deuxième couche. La syntaxe servant à décrire les transitions est basée sur deux principes. Premièrement, la flèche (`->`) est un nom de méthode valide dans le langage Scala. Deuxièmement, le point servant à construire les appels de méthodes sans arguments peut être remplacé par un espace suivi de l'argument sans parenthèses (`obj mth d` $\Longleftrightarrow$ `obj.mth(d)`).

Optimisations Nous avons mis en œuvre deux alternatives pour la deuxième couche. Ces implémentations fournissent la même sémantique que l'implémentation basée sur des *join patterns*, mais elles exploitent les connaissances du domaine des automates finis. La première, `FSM_N` dispose d'une variable pour l'état actuel. Elle implémente la fonction de transition par l'utilisation de la fonction `HashMap` du langage Scala. La deuxième optimisation est fondée sur le même principe, mais elle énumère les états et les événements et utilise un tableau à deux dimensions pour la fonction de transition.

C.5.4.1 Automates alternatifs

Notre langage dédié peut seulement décrire des automates finis sans arguments. Nous avons toutefois montré que des arguments sont utiles à la fois pour les actions et les états. Par exemple, dans le cas d'un verrouillage en lecture/écriture, un état paramétré permet de représenter plusieurs états.

C.6 Moniteur d'événements

Les gestionnaires de JEScala sont des fonctions « impures », susceptibles de manipuler des données mutables, d'où un risque de compétition problématique. La machine virtuelle Java utilisée par Scala fournit des moniteurs qui peuvent être utilisés pour synchroniser les gestionnaires. L'utilisation de ces moniteurs pose toutefois des problèmes. Par exemple, tous les gestionnaires qui s'excluent mutuellement doivent utiliser le même moniteur, ce qui peut provoquer des erreurs lors de l'utilisation de l'enregistrement dynamique. Un autre inconvénient est que la combinaison de l'enregistrement dynamique avec la synchronisation des gestionnaires peut introduire des interblocages.
Nous introduisons un moniteur d'événements qui applique l'exclusion mutuelle parmi les gestionnaires qui sont exécutées de façon asynchrone par rapport à l'événement. Le moniteur d'événements synchronise les occurrences d'événements qui donnent lieu à l'exécution d'un gestionnaire qui fait partie d'un groupe de gestionnaires.
Un moniteur d'événements fournit à l'utilisateur un événement servant à enregistrer un gestionnaire, lorsque celui-ci doit observer un événement. L'événement du moniteur d'événements est synchrone, car il permet au moniteur d'événements de savoir quand tous ses gestionnaires retournent. L'événement d'origine est observé de façon asynchrone. Si cet événement est synchrone l'opérateur double bang (`!!`) le convertit en

événement asynchrone.

C.7 Implémentation

C.7.1 Graphe d'événements

Tout d'abord nous expliquons que les expressions d'événements décrivent des graphes orientés acycliques par lesquels les occurrences d'événements sont transmises. Le graphe d'événements peut arrêter la propagation d'un événement lorsqu'il n'y a pas de récepteur. Par conséquent, l'enregistrement d'un gestionnaire peut activer et désactiver les parties du graphe. Chaque fois qu'un événement primitif est déclenché, une traversée du graphe recueille tous les gestionnaires directs et indirects de l'événement.

C.7.2 Événements synchrones et asynchrones

Lorsqu'un thread déclenche un événement primitif synchrone, c'est ce même thread qui recueille les gestionnaires associés et les exécute. Pour les événements asynchrones, les gestionnaires sont exécutés par un autre thread. Au cours du parcours récursif du graphe d'événements, quatre paramètres sont transmis. Le premier paramètre est un ID unique pour chaque occurrence d'événement capable de détecter des boucles dans un graphe d'événements. Le deuxième paramètre est l'argument de l'événement. Un paramètre de type booléen indique si l'événement déclenché est synchrone. Le dernier argument est une collection mutable à laquelle chaque nœud dans le graphe ajoute ses gestionnaires enregistrés. Ces gestionnaires sont des fonctions partiellement appliquées sans argument.

C.7.3 Disjonction

Une disjonction dispose d'une file d'attente (angl. : *event queue*) pour chaque événement qui participe à un de ses *join patterns*. Chaque élément dans une file d'attente d'événements contient l'argument d'une occurrence d'événement reçue. Pour les événements synchrones elle contient également le thread émetteur bloqué. Chaque fois que l'occurrence d'un événement est captée, des tests associés à la disjonction recherchent le *join pattern* correspondant en se basant sur l'occurrence la plus ancienne dans la file d'attente correspondante.

C.7.3.1 Optimisations

Pool de threads Les disjonctions disposent d'un pool de threads pour exécuter dans un nouveau thread les gestionnaires d'un événement asynchrone primitif.

La collecte asynchrone des gestionnaires Puisque la collecte des gestionnaires ne prend que peu de temps, tous les événements primitifs utilisent le thread qui les a déclenchés. Ceci permet d'éviter, dans le cas d'un événement asynchrone, qui requiert l'utilisation d'un nouveau thread pour exécuter ses gestionnaires, de faire appel à ce nouveau thread dans le cas où aucun gestionnaire n'a été trouvé pendant la phase de collecte.

Disjonction solitaire La mise en attente (angl. : *enqueueing*) d'un événement reçu est très rapide. Lorsqu'un événement asynchrone est observé uniquement par des *join patterns* dans une disjonction, il est possible de mettre cet événement en attente pendant la collecte des gestionnaires. Ceci évite l'exécution de gestionnaires par un autre thread.

Remplacer la file d'attente par un compteur Lorsqu'un événement dans une disjonction est toujours asynchrone, son thread n'est pas stocké dans sa file d'attente associée. Quand un événement n'a pas d'arguments, on peut remplacer sa file d'attente par un compteur d'éléments en attente. La mise en attente d'un élément se traduit par l'incrémentation du compteur et vice-versa. Les opérations de comptage sont plus efficaces que celles sur une file d'attente. C'est au programmeur de mettre en place cette optimisation quand il sait que les occurrences d'un événement seront toujours asynchrones. Par sécurité, une occurrence synchrone résulte en une exception. En effet, dans ce cas, le compteur qui remplace la file d'attente d'événement n'est pas en mesure de stocker le thread de l'émetteur d'une occurrence d'événement.

C.7.4 Langage dédié

La mise en œuvre du langage dédié est constituée de deux couches. Le trait `FSM` est la première couche. Il collecte toutes les transitions et expose ses informations à une sous-classe qui est responsable de la création des structures pour chacune des trois implémentations, à savoir celle basée sur les *join patterns* (`FSM_J`), une table de hachage (`FSM_N`) ou un tableau (Array) Scala (`FSM_E`). Le trait `FSM` fournit les méthodes flèche (`->`), `on`, `apply` et `trigger` qui sont toutes des méthodes intermédiaires de mise en mémoire-tampon. Le tampon intermédiaire est ajouté à la collection de transitions quand les trois arguments principaux (la source, la destination et l'action) sont connus. Au lieu d'ajouter ces méthodes à tous les états, elles sont mises en œuvre dans l'adaptateur (angl. : *wrapper*) `Arrow`. L'état (`S`) est donc reconverti implicitement en adaptateur `Arrow`.
Pour supprimer une transition, ses trois principaux arguments doivent également être spécifiés. La méthode abstraite `initialState` de trait *FSM* est mise en œuvre par une sous-classe qui génère les structures de données pour l'implémentation souhaitée.
Contrairement aux disjonctions de JEScala, le langage dédié prend en charge l'héritage. Pour la mise en œuvre basée sur les *join patterns* (`FSM_J`), il faut désactiver la disjonc-

tion générée par la super-classe en la remplaçant par une disjonction nouvelle dans la sous-classe.
Nous avons créé des versions optimisées pour remplacer le trait à base de *join patterns*. La première optimisation (`FSM_N`) utilise la mise en attente pour les événements d'action. Si plusieurs transitions sont possibles, le choix est aléatoire. L'état est représenté par une variable qui est seulement accessible aux blocs synchronisés. La deuxième optimisation (`FSM_E`) est fondée sur la première, mais elle utilise un tableau à deux dimensions au lieu d'une instance de la classe `HashMap` de Scala pour la fonction de transition.

C.7.5 Moniteur d'événements

D'abord nous expliquons l'implémentation flexible d'un moniteur d'événements pour un groupe d'événements et ses gestionnaires fixes. Un objet dédié fournit un événement connexe synchrone pour chaque événement. Chaque événement connexe est déclenché par un gestionnaire connexe défini dans l'objet dédié et enveloppé dans un bloc protégé par le mot-clef `synchronized`, utilisant ainsi le moniteur sous-jacent à l'objet dédié qui rend mutuellement exclusifs les gestionnaires connexes. Au lieu d'enregistrer les gestionnaires avec les événements originaux, ils sont enregistrés avec les événements connexes, alors que ce sont les gestionnaires connexes qui sont enregistrés avec les événements originaux. Cette implémentation est rigide, parce que son code change quand on ajoute un événement ou un gestionnaire.
La classe `Synchronizer` hérite de la classe `Actor` de Scala : un synchroniseur, instance de `Synchronizer` est un acteur. Sa méthode `getProxyEvent` fournit un événement connexe impératif synchrone pour chaque événement observé. Les événements observés ne sont pas bloquants car ils sont gérés par un gestionnaire dédié qui envoie un message d'acteur asynchrone pour chaque occurrence d'événement observé. L'acteur déclenche impérativement l'événement connexe avec le paramètre reçu via le message. Le thread unique de l'acteur attend jusqu'à ce que tous les gestionnaires d'événement synchrone aient terminé leur exécution.
Pour restaurer le type de l'argument de l'événement observé, chaque événement est numéroté séquentiellement par le synchroniseur. Ces numéros constituent les clés de deux tableaux. Le premier tableau contient les gestionnaires qui fournissent le paramètre d'occurrence d'événement et le numéro de séquence pour l'acteur. Quand l'acteur reçoit ces données, il consulte le deuxième tableau pour rechercher la fonction servant à déclencher de façon impérative l'événement synchrone connexe.

C.8 Validation

C.8.1 Évaluation statique

Nous avons effectué plus de trente petites études de cas synthétiques. Nous avons mis en œuvre chacune d'elles à l'aide du langage JEScala et un sous-ensemble limité (JL) de celui-ci qui n'exploite pas le système d'événements avancé. Nous avons comparé les métriques suivantes : le nombre de gestionnaires, le nombre d'événements déclenchés impérativement, le nombre total d'événements et le nombre de lignes de code. Comparée à la version JL, la version JEScala utilise 54 % de gestionnaires en moins. Nous avons constaté également une réduction de 61 % dans les déclenchements d'événements explicites. De moindre importance, nous avons constaté que le nombre d'événements a diminué de 2 % et nous avons observé une réduction de 8 % dans le nombre des lignes de code.

C.8.2 Évaluation dynamique

L'efficacité n'était pas l'objectif principal de notre mise en œuvre. Toutefois, l'implémentation de JEScala est comparable à d'autres implémentations basées sur la JVM (Esper [35], ScalaJoins [51]), à Cω [106] et JoCaml [23].
Nous avons créé un point de référence pour les *join patterns* en comparant JEScala à d'autres langages de *joins*. Le nombre de *join patterns* par seconde diminue lorsque la disjonction passe de deux *join patterns* binaires à trois *join patterns* de trois événements. La performance ne diminue pas plus que dans d'autres langages de *joins* quand on dépasse le nombre de trois paramètres.
Nous avons aussi mesuré l'effet des optimisations d'événements asynchrones qui peuvent être activés dans JEScala. Premièrement, l'utilisation d'un pool de threads pour exécuter les gestionnaires d'événements asynchrones a l'effet le plus important. Deuxièmement, l'optimisation des disjonctions solitaires évite l'exécution asynchrone de gestionnaires. Finalement, le remplacement de la mise en place d'une mémoire-tampon pour les attentes d'événements par un compteur d'événements asynchrones sans argument n'a qu'un effet réduit.

C.8.3 Les automates finis

Nous avons d'abord montré d'abord que le langage dédié permet de réduire le nombre de lignes de code. La plupart des lignes sont enregistrées pour des automates finis avec un nombre d'états important.
Ensuite, nous avons considéré un programme test fondé sur un automate fini avec trois états qui sont connectés de façon triangulaire. Chaque état est capable de gérer deux événements d'action : dans le sens des aiguilles d'une montre et vice-versa. Ces actions sont

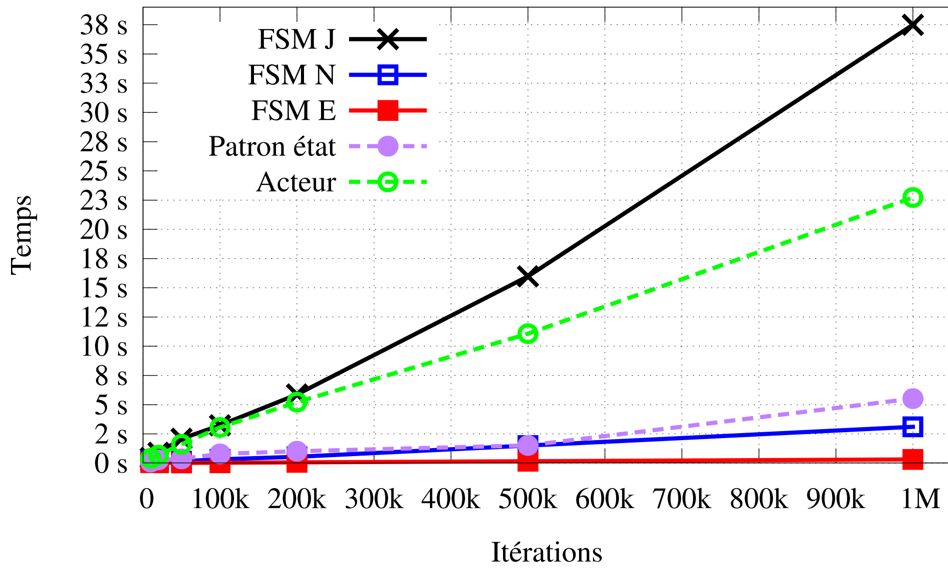


FIGURE C.12 – Temps d'exécution pour le benchmark du triangle.

générées par un générateur pseudo-aléatoire basé sur la suite de Fibonacci avec une valeur de départ fixe. La figure C.12 montre que l'implémentation à base de *join patterns*, qui peut traiter des événements synchrones (patron état) et des événements asynchrones (acteur), est la plus lente. La mise en œuvre basée sur une table de hachage (`FSM_N`) est dix fois plus rapide, tandis que la mise en œuvre basée sur un tableau (`FSM_E`) est la plus efficace, puisqu'elle augmente encore la vitesse d'exécution davantage, à savoir par un facteur dix.

C.8.4 Moniteurs d'événements

Nous avons utilisé des éléments extraits d'un univers basé sur les événements séquentiels, une étude de cas qui a été misw en œuvre en EScala. Nous avons supprimé tous les effets de retardement et le code qui affiche la grille après chaque étape. Ensuite, nous avons mesuré le temps requis pour effectuer un certain nombre de pas de simulations dans la mise en œuvre en EScala et la mise en œuvre en JEScala, qui utilise un moniteur d'événements pour introduire la concurrence. La figure C.13 montre que la vitesse la version concurrente, construite simplement à partie de la version séquentielle, progresse effectivement bien plus vite que cette dernière.

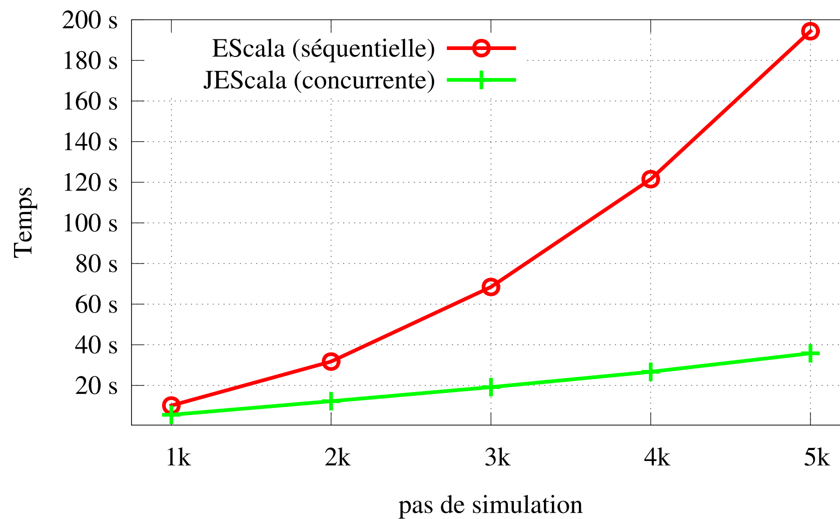


FIGURE C.13 – Simulation d'univers, version séquentielle et concurrente.

C.9 Les travaux futurs

Nos principales pistes de travaux sur JEScala sont les suivantes :

— l'exploration des fondations théoriques de JEScala ;

— l'amélioration de son implémentation du point de vue de l'efficacité et du confort de l'utilisateur en complétant l'implémentation sous la forme d'une bibliothèque par un compilateur qui met en œuvre des analyses de programmes ainsi que la sytaxe native de JEScala ;

— l'extension de JEScala afin de prendre en compte d'autres styles de programmation, par exemple l'ajout de fenêtres de temps pour traiter des événements complexes (angl. CEP), la prise en compte d'événements avec retour pour autoriser des formes classiques de programmation par aspects et le traitement de la programmation réactive.

Bibliography

[1] P.A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 313–321. IEEE Computer Society, July 1996. 77

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. 54, 197

[3] *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011*. ACM, March 2011. 220, 222, 224

[4] Ivica Aracic, Vaidas Gasiūnas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer Verlag, 2006. 34

[5] Joe Armstrong. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, June 2007. 54, 69, 198

[6] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010. 20, 54, 189

[7] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59. ACM, August 1977. 20, 54, 57, 189, 196

[8] Hendrik Pieter Barendregt. *The lambda calculus*. North-Holland Amsterdam, 1984. 58

[9] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, September 2004. 21, 65, 66, 71, 82, 101, 155, 171, 189, 190, 198

[10] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, January 1985. 58

[11] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM. 59

[12] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In Shail Arora and Gary T. Leavens, editors, *OOPSLA '09*, pages 117–136. ACM, October 2009. 70, 171

[13] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 83–92, New York, NY, USA, March 2004. ACM. 33

[14] Christoph Bockisch, Somayeh Malakuti, Mehmet Akşit, and Shmuel Katz. Making aspects natural: events and composition. In AOSD2011 [3], pages 285–300. 173

[15] Eric Bodden, Éric Tanter, and Milton Inostroza. Joint point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology*, 2014. To appear. 27, 172

[16] Jonas Bonér. What are the key issues for commercial AOP use: How does AspectWerkz address them? In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 5–6, New York, NY, USA, 2004. ACM. 34

[17] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computer Survey*, 30(3):291–329, September 1998. 55

[18] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computer Survey*, 27(1):63–107, March 1995. 53

[19] Dick Buttlar and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996. 53

[20] Denis Caromel, Luis Mateu, Guillaume Pothier, and Éric Tanter. Parallel object monitors. *Concurrency and Computation: Practice and Experience*, 20(12):1387–1417, July 2008. 27, 173

[21] Denis Caromel, Luis Mateu, and Éric Tanter. Sequential object monitors. In *ECOOP '04*, volume 3086 of *Lecture Notes in Computer Science*, pages 316–340. Springer Verlag, 2004. 27, 132, 173

[22] Siobhán Clarke and Elisa Baniassad. *Aspect-oriented analysis and design*. Addison-Wesley Professional, 2005. 33

[23] Sylvain Conchon and Fabrice Le Fessant. JoCaml: mobile agents for objective-caml. In *ASAMA '99*, pages 22–29. IEEE Computer Society, 1999. 21, 63, 71, 189, 190, 198, 216

[24] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer Verlag, 2006. 177

[25] B.J. Cox. *Object oriented programming*. Addison-Wesley, Reading, MA, Jan 1985. 32

[26] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):14:1–14:60, May 2008. 34

[27] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007*, volume 7, pages 412–422, 2007. 45, 195

[28] Digia. Qt cross platform application framework. http://qt–project. org, 2014. 42, 43

[29] Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971. 20, 52, 183, 188, 196

[30] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 79–88, New York, NY, USA, 2006. ACM. 77

[31] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186. Springer Verlag, 2001. 77

[32] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, December 2002. 34

[33] Rémi Douence and Mario Südholt. Event-based AOP. http://www. emn. fr/ z-info/ eaop/, 2006. 77

[34] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In Lopes and Fisher [76], pages 391–406. 176

[35] EsperTech. Espertech company website. http://www.espertech.com. 45, 47, 154, 173, 195, 216

[36] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computer Survey*, 35(2):114–131, June 2003. 38, 48, 50, 111

[37] Patrick Th. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, pages 570–594, 2009. 46, 195

[38] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced separation of Concerns, OOPSLA*, volume 2000, October 2000. 33

[39] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, January 1996. ACM. 20, 32, 58, 59, 63, 183, 188, 189, 197

[40] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join calculus. In *Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *FST TCS 2000*, pages 397–408, London, UK, UK, 2000. Springer Verlag. 171, 172, 176

[41] Daniel P. Friedman and David Stephen Wise. The impact of applicative programming on multiprocessing. In Philip H. Enslow, editor, *Proceedings of the 1976 International Conference on Parallel Processing*, pages 269–272. IEEE Computer Society, August 1976. 57

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., 1994. 39, 108, 111, 207

[43] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I: Conference Contributions - Volume I*, volume 551 of *Lecture Notes in Computer Science*, pages 31–44, London, UK, UK, 1991. Springer Verlag. 32, 38, 91, 172, 194

[44] Vaidas Gasiūnas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular event-driven object interactions in Scala. In AOSD2011 [3], pages 227–240. 23, 32, 90, 93, 139, 166, 176, 184, 191, 202

[45] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. 39

[46] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996. 25, 34, 53, 193

[47] Munish K. Gupta. *Akka Essentials*. Packt Publishing Ltd, October 2012. 73

[48] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2 3):202–220, 2009. Distributed Computing Techniques. 55

[49] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In Theo D'Hondt, editor, *ECOOP 2010 Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer Verlag, 2010. 55

[50] Philipp Haller and Frank Sommers. *Actors in Scala*. Artima Inc, January 2012. 54, 55, 73

[51] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, volume 5052 of *COORDINATION'08*, pages 135–152, Berlin, Heidelberg, June 2008. Springer Verlag. 21, 68, 82, 170, 189, 190, 216

[52] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Trans. Software Eng.*, 1(2):199–207, March 1975. 20, 26, 53, 183, 184, 188, 196

[53] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003. 42, 53, 66

[54] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. 20, 57, 189, 197

[55] Carl Hewitt. How to use what you know. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, pages 189–198, Tbilisi, Georgia, 1975. Morgan Kaufmann Publishers Inc. 54

[56] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, August 1973. Morgan Kaufmann Publishers Inc. 20, 28, 54, 172, 188, 196, 197

[57] Peter Hibbard. Parallel processing facilities. *New Directions in Algorithmic Languages*, pages 1–7, 1976. 57

[58] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974. 20, 26, 53, 183, 184, 188, 196

[59] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 58

[60] G. Stewart Itzstein and Mark Jasiunas. On implementing high level concurrency in Java. In *Advances in Computer Systems Architecture*, volume 2823 of *Lecture*

Notes in Computer Science, pages 151–165. Springer Verlag, 2003. 21, 62, 65, 71, 171, 189, 190, 198

[61] F. James. A review of pseudorandom number generators. *Computer Physics Communications*, 60(3):329–344, 1990. 165

[62] AOP JBoss. Jboss AOP-Aspect-Oriented framework for Java. http://docs. jboss. org/aop/1.1/aspect-framework/reference/en/html/index. html, 2004. 34

[63] The JEScala site. http://www.stg.tu-darmstadt.de/research, 2014. 25, 152

[64] Simon L. Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 53

[65] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In AOSD2011 [3], pages 253–264. 46, 195

[66] Gregor Kiczales. *The art of the metaobject protocol*. MIT press, 1991. 33

[67] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and WG Griswold. An overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, pages 327–353, 2001. 32, 33, 34, 193

[68] 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. 22, 23, 32, 33, 113, 184, 190, 191

[69] Glenn E. Krasner and Stephen T. Pope. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of object-oriented programming*, 1(3):26–49, 1988. 39

[70] Ralf Lämmel and Kris De Schutter. What does aspect-oriented programming mean to Cobol? In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 99–110, New York, NY, USA, 2005. ACM. 34

[71] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964. 59

[72] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison Wesley, second edition, 1999. 108

[73] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 3.12): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, July 2011. 53

[74] Jean-Jacques Lévy. Some results in the join-calculus. In *Theoretical Aspects of Computer Software*, volume 1281 of *TACS '97*, pages 233–249. Springer Verlag, 1997. 59

[75] Yuheng Long, Sean L. Mooney, Tyler Sondag, and Hridesh Rajan. Implicit invocation meets safe, implicit concurrency. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE'10)*, pages 63–72. ACM, 2010. 27, 77, 172

[76] Cristina Videira Lopes and Kathleen Fisher, editors. *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. ACM, 2011. 221, 228

[77] David C. Luckham. *The power of events - an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley Professional, October 2013. 27, 39, 44, 194

[78] Louis Mandel and Luc Maranget. Programming in JoCaml — extended version. Technical Report 6261, MOSCOVA - INRIA Rocquencourt, 2007. https://www.lri.fr/ mandel/papiers/MandelMaranget-RR-2007.pdf. 158

[79] Louis Mandel and Luc Maranget. *The JoCaml language - Documentation and user's manual*. Inria, August 2012. Release 4.00. 21, 64, 82, 190

[80] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A fine-grained join point model for more reusable aspects. In Naoki Kobayashi, editor, *Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147. Springer Verlag, 2006. 34

[81] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer Verlag, 2003. 34, 37

[82] Chris Metz. IP QoS: Traveling in first class on the internet. *IEEE Internet Computing*, 3(2):84–88, 1999. 80

[83] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for AJAX applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM. 119

[84] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. 58

[85] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I. *Information and computation*, 100(1):1–40, 1992. 58, 59

[86] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992. 18, 183, 187, 196

[87] Taketosh Nishimori and Yasushi Kuno. Mogemoge: A Programming Language Based on Join Tokens. In *Proceedings of The International Workshop on Information Science Education & Programming Languages, Korean University & University of Tsukuba 2006*, pages 22–27, January 2006. 27, 171

[88] Angel Núñez, Jacques Noyé, Vaidas Gasiūnas, and Mira Mezini. *Aspect-Oriented, Model-Driven Software Product Lines - The AMPLE Way*, chapter Product Line Implementation with ECaesarJ. Cambridge University Press, 2011. 90, 125, 202

[89] Martin Odersky. Functional nets. In Gert Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 1–25. Springer Verlag, 2000. 64, 65

[90] Martin Odersky. An introduction to functional nets. In *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 333–377. Springer Verlag, 2002. 21, 64, 171, 189, 190, 198

[91] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2nd edition, 2010. 21, 25, 32, 53, 57, 197

[92] Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional separation of concerns for java. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 734–737, New York, NY, USA, June 2000. ACM. 33, 193

[93] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings USENIX Summer Conference*, pages 247–256. USENIX, June 1990. 18, 187

[94] Ignacio Solla Paula. JCThorn: Extending Thorn with joins and chords. Master's thesis, Department of Computing, Imperial College London, 2010. 21, 70, 171, 189

[95] Thomas Pawlitzki and Friedrich Steimann. Implicit invocation of traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2085–2089. ACM, 2010. 27, 172

[96] James Lyle Peterson and Abraham Silberschatz. *Operating system concepts*, volume 2. Addison-Wesley Reading, MA, 1985. 18

[97] Hubert Plociniczak and Susan Eisenbach. JErlang: Erlang with joins. In *COORDINATION '10*, volume 6116 of *Lecture Notes in Computer Science*, pages 61–75. Springer Verlag, 2010. 21, 60, 62, 69, 171, 189, 198

[98] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: Efficient dynamic weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 100–109, New York, NY, USA, 2003. ACM. 33

[99] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 141–147, New York, NY, USA, 2002. ACM. 33

[100] FFmpeg project. Ffmpeg website. https://www.ffmpeg.org/. 161

[101] Hridesh Rajan, Steven M. Kautz, Eric Line, Sarah Kabala, Ganesha Upadhyaya, Yuheng Long, Rex Fernando, and Loránd Szakács. Capsule-oriented programming. Technical Report 13-01, Iowa State U., Computer Sc., 2013. 172

[102] Hridesh Rajan, Steven M. Kautz, and Wayne Rowcliffe. Concurrency by modularity: design patterns, a case in point. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *OOPSLA '10*, pages 790–805. ACM, 2010. 172

[103] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, volume 5142 of *ECOOP '08*, pages 155–179, Berlin, Heidelberg, July 2008. Springer Verlag. 34, 46, 172, 195

[104] Awais Rashid and Ana Moreira. Domain models are not aspect free. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 155–169. Springer Verlag, 2006. 33

[105] Tim Rentsch. Object oriented programming. *SIGPLAN Notices*, 17(9):51–57, September 1982. 32

[106] Microsoft Research. Cω website. http://research.microsoft.com/en-us/um/cambridge/projects/comega/. 66, 171, 198, 216

[107] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3-4):233–247, 1993. 63

[108] Claudio V. Russo. The joins concurrency library. In *PADL '07*, volume 4354 of *Lecture Notes in Computer Science*, pages 260–274. Springer Verlag, 2007. 21, 66, 171, 189, 198

[109] Claudio V. Russo. Join patterns for Visual Basic. In Gail E. Harris, editor, *OOPSLA '08*, pages 53–72. ACM, October 2008. 67, 171, 198

[110] Douglas C. Schmidt. The adaptive communication environment. *DOC group, Washington University*, 1994. 53

[111] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc. 33

[112] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In Jan Vitek, editor, *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008)*, pages 104–128, Berlin, Heidelberg, 2008. Springer Verlag. 55

[113] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 481–497, New York, NY, USA, 2006. ACM. 33

[114] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1):1:1–1:43, July 2010. 46, 77, 195

[115] Maximilian Stoerzer and Stefan Hanenberg. A classification of pointcut language constructs. In *Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT) held in conjunction with AOSD*, 2005. 34

[116] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the second International Conference on Aspect-oriented Software Development*, AOSD '03, pages 21–29, New York, NY, USA, 2003. ACM. 46, 195

[117] Nicolas Tabareau. A theory of distributed aspects. In Jean-Marc Jézéquel and Mario Südholt, editors, *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 133–144. ACM, 2010. 172

[118] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 27–46, New York, NY, USA, 2003. ACM. 34

[119] AspectJ Team. The AspectJ programming guide. version 1.5.3, 2006. 35

[120] Rapide Design Team. Guide to the Rapide 1.0 language reference manuals. *Computer System Labs Stanford University, draft edition*, 1997. 46, 195

[121] Mads Torgersen. Querying in C#: how language integrated query (LINQ) works. In *OOPSLA 2007, Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, October 2007. 66, 198

[122] Aaron J. Turon and Claudio V. Russo. Scalable join patterns. In Lopes and Fisher [76], pages 575–594. 67, 171, 176, 198

[123] Peter van Eijk and Michel Diaz. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., January 1989. 58

[124] Jurgen M. Van Ham. Adding high-level concurrency to EScala. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, pages 19–20, New York, NY, USA, 2012. ACM. 27

[125] Jurgen M. Van Ham, Guido Salvaneschi, Mira Mezini, and Jacques Noyé. JEScala: Modular coordination with declarative events and joins. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 205–216, New York, NY, USA, 2014. ACM. 27

[126] Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A type system for data-centric synchronization. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 304–328, Berlin, Heidelberg, 2010. Springer Verlag. 20, 189

[127] Craig Walls and Ryan Breidenbach. *Spring in Action*. Manning Publications Co., Greenwich, CT, USA, 2007. 34

[128] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004. 34

[129] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM. 45, 195

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

TECHNISCHE UNIVERSITÄT DARMSTADT

MINES Nantes

Thèse de Doctorat

Jurgen M. VAN HAM

Intégration de la programmation concurrente à la programmation par objets, aspects et événements

Seamless Concurrent Programming of Objects, Aspects and Events

Résumé
L'utilisation de concepts avancés de programmation concurrente permet de dépasser les inconvénients de l'utilisation de techniques de bas niveau à base de verrous ou de moniteurs. Elle augmente le niveau d'abstraction, libérant les programmeurs d'applications concurrentes d'une focalisation excessive sur des détails. Cependant, avec les approches actuelles, la logique nécessaire à la mise en place de schémas de coordinations complexes est fragmentée en plusieurs points de l'application sous forme de « join patterns », de notifications et de la logique applicative qui crée implicitement des dépendances entre les canaux de communication et donc, indirectement, les « join patterns » (qui définissent ces canaux). Nous présentons JEScala, un langage qui capture les schémas de coordination (d'une application concurrente) d'une manière plus expressive et modulaire, en s'appuyant sur l'intégration fine d'un système d'évènements avancé et des « join patterns ». Nous implémentons des automates finis à partir de « joins » à l'aide de JEScala et introduisons un langage dédié à la définition de ces automates finis permettant d'en obtenir des implémentations plus efficaces. Nous validons notre approche avec des études de cas et évaluons l'efficacité de son exécution. Nous comparons la performance de trois implémentations d'un automate fini. Nous validons enfin l'idée d'un moniteur d'évènements en créant un programme JEScala concurrent à partir d'un découpage d'un programme séquentiel.

Mots clés
Programmation par événements, programmation par aspects, concurrence, « Join Patterns », Scala

Abstract
The advanced concurrency abstractions provided by the Join calculus overcome the drawbacks of low-level techniques such as locks and monitors. They rise the level of abstraction, freeing programmers that implement concurrent applications from the burden of concentrating on low-level details. However, with current approaches the coordination logic involved in complex coordination schemas is fragmented into several pieces including join patterns, data emissions triggered in different places of the application, and the application logic that implicitly creates dependencies among channels, hence indirectly among join patterns. We present JEScala, a language that captures coordination schemas in a more expressive and modular way by leveraging a seamless integration of an advanced event system with join abstractions. We implement Joins-based state machines using JEScala and introduce a domain specific language for finite state machines that make faster alternative implementations possible. We validate our approach with case studies and we provide a first performance assessment. We compare the performance of three different implementations of a finite state machine. Finally, we validate the idea of constructing a concurrent JEScala program by using the parts of a sequential Event-Based program in combination with an event monitor, a component that synchronizes handling of multiple events.

Keywords
Event-driven Programming, Aspect-Oriented Programming, Concurrency, Join Patterns, Scala

L'UNIVERSITÉ NANTES ANGERS LE MANS