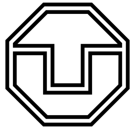


Faculty of Computer Science Institute of Software- and Multimedia-Technology

Software Technology Group

A PURE EMBEDDING OF ROLES

EXPLORING 4-DIMENSIONAL DISPATCH FOR ROLES IN STRUCTURED CONTEXTS

Max Leuthäuser

Born on: 20th April 1989 in Dresden

DISSERTATION

to achieve the academic degree

DOKTOR-INGENIEUR (DR.-ING.)

Referee

Prof. Anthony Sloane

Supervising professors

Prof. Uwe Aßmann
Prof. Christel Baier

Submitted on: 31st May 2017

Und es liegt darin der Wille, der nicht stirbt. Wer kennt die Geheimnisse des Willens und seine Gewalt? Denn Gott ist nichts als ein großer Wille, der mit der ihm eignen Kraft alle Dinge durchdringt. Der Mensch überliefert sich den Engeln oder dem Nichts einzig durch die Schwäche seines schlaffen Willens.

Joseph Glanvill

STATEMENT OF AUTHORSHIP

I hereby certify that I have authored this Dissertation entitled *A Pure Embedding of Roles* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 31st May 2017

Max Leuthäuser

ABSTRACT

Present-day software systems have to fulfill an increasing number of requirements, which makes them more and more complex. Many systems need to anticipate changing contexts or need to adapt to changing business rules or requirements. The challenge of 21th-century software development will be to cope with these aspects. We believe that the role concept offers a simple way to adapt an object-oriented program to its changing context. In a role-based application, an object plays multiple roles during its lifetime. If the contexts are represented as first-class entities, they provide dynamic views to the object-oriented program, and if a context changes, the dynamic views can be switched easily, and the software system adapts automatically. However, the concepts of roles and dynamic contexts have been discussed for a long time in many areas of computer science. So far, their employment in an existing object-oriented language requires a specific runtime environment. Also, classical object-oriented languages and their runtime systems are not able to cope with essential role-specific features, such as true delegation or dynamic binding of roles. In addition to that, contexts and views seem to be important in software development. The traditional code-oriented approach to software engineering becomes less and less satisfactory. The support for multiple views of a software system scales much better to the needs of today's systems. However, it relies on programming languages to provide roles for the construction of views.

As a solution, this thesis presents an implementation pattern for role-playing objects that does not require a specific runtime system, the *SCala ROles Language* (*SCROLL*). Via this library approach, roles are embedded in a statically typed base language as dynamically evolving objects. The approach is pure in the sense that there is no need for an additional compiler or tooling. The implementation pattern is demonstrated on the basis of the Scala language. As technical support from Scala, the pattern requires dynamic mixins, compiler-translated function calls, and implicit conversions. The details how roles are implemented are hidden in a Scala library and therefore transparent to *SCROLL* programmers. The *SCROLL* library supports roles embedded in structured contexts. Additionally, a four-dimensional, context-aware dispatch at runtime is presented. It overcomes the subtle ambiguities introduced with the rich semantics of role-playing objects. *SCROLL* is written in Scala, which blends a modern object-oriented with a functional programming language. The size of the library is below 1400 lines of code so that it can be considered to have minimalistic design and to be easy to maintain. Our approach solves several practical problems arising in the area of dynamical extensibility and adaptation.

ACKNOWLEDGMENTS

We have talked so much about the reader, but you cannot forget that the opening line is important to the writer, too. To the person who is actually boots-on-the-ground. Because it is not just the readers way in, it is the writers way in also, and you have got to find a doorway that fits us both.

Stephen King

First, I would like to express my special thanks to my supervisor Uwe Aßmann. He supported me and the whole research group with a very special research atmosphere and gave me many useful and path-leading ideas, hints and comments. Additionally, I am very thankful for providing me the opportunity for my stay abroad at the Macquarie University in Sydney. Also, his support allowing me to attend various conferences and workshops was very much appreciated.

Special thanks to Sebastian Richly and Sebastian Götz for initially introducing me to the very specific topic of role-based programming. The same goes to Wolfgang Lehner, who does a great job leading the RoSI research training group, and who provided me with many fruitful discussions and comments on my work.

Moreover, I like to thank all the past and current colleagues for the relaxing and interesting research atmosphere at our chair. Off-topic chats as well as in-depth discussions about ongoing papers or my thesis were always a pleasure. As a member of the RoSI research group special thanks to Thomas, Tobias, Johannes, Steffen, and Stephan for their helpful comments and hints.

Furthermore, a special thanks to Anthony Sloane for hosting me at the Macquarie University in Sydney, introducing the city to me, showing me the local highlights and providing interesting discussions about reference attribute grammars and beyond.

I am also thankful to my family and friends. My parents always supported me with help, where help was needed. Thank you, Fabian, Uwe, Ronny, Felix, and Martin for all the off-topic chats and fun we had during our student times.

A very special thanks goes to my girlfriend Anne, who greatly supported me during my entire PhD project. Even if she has no special interest in computer science, she always encouraged me to keep going. So I did.

PUBLICATIONS

Sebastian Götz, Max Leuthäuser, Christian Piechnick, Jan Reimann, Sebastian Richly, Julia Schroeter, Claas Wilke, and Uwe Aßmann. Entwicklung Cyber-Physikalischer Systeme am Beispiel des NAO-Roboters. *Chemnitzer Linux-Tage - Tagungsband*, 2012

Sebastian Götz, Max Leuthäuser, Jan Reimann, Julia Schroeter, Christian Wende, Claas Wilke, and Uwe Aßmann. A Role-Based Language for Collaborative Robot Applications. In *Leveraging Applications of Formal Methods, Verification, and Validation*, ISoLA SARS, pages 1–15. Springer, 2011

Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 141–160. Springer International Publishing, 2014. ISBN 978-3-319-11244-2. doi: 10.1007/978-3-319-11245-9_8

Max Leuthäuser. SCROLL - A Scala-based library for Roles at Runtime. In van der Storm, Tijs and Erdweg, Sebastian, editor, *Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015)*, volume abs/1508.03536, pages 7–8. van der Storm, Tijs and Erdweg, Sebastian, 2015

Max Leuthäuser and Uwe Aßmann. Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Establishing View-based Programming. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, MORSE/VAO '15, pages 25–33, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3614-7. doi: 10.1145/2802059.2802062

Tobias Jäkel, Martin Weißbach, Kai Herrmann, Hannes Voigt, and Max Leuthäuser. Position Paper: Runtime Model for Role-Based Software Systems. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 380–387, July 2016. doi: 10.1109/ICAC.2016.17

Max Leuthäuser. Pure Embedding of Evolving Objects. In *Proceedings of ADAPTIVE 2017, The Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications*, ADAPTIVE 2017. IARIA, 2017

CONTENTS

Statement of authorship 5

Abstract 7

Acknowledgments 9

Publications 11

Contents 13

I. Introduction 15

1. Thesis Topic 17

2. Contributions and Outline 19

II. Background and Problem Analysis 21

3. Foundations of Roles 23
   - 3.1. The Historical Term Role 23
   - 3.2. Issues of Object-oriented Software Design 24
   - 3.3. The Introduction of Roles 25
   - 3.4. Towards a Common Understanding of Roles 27
   - 3.5. The Compartment Role Object Model (CROM) 30
   - 3.6. The Automaton-based Role-binding Process 33
   - 3.7. Runtime-specific Role Features 34

4. Foundations of Dispatch 39
   - 4.1. Predicate Dispatch 41
   - 4.2. Multi-dimensional Dispatch 41

5. Foundations of Graphs and Graph Filtering 43

6. A Motivating Example for Multi-dimensional Dispatch 47

7. Research Challenges 51

III. Roles in Structured Contexts at Runtime 53

8. The Embedded DSL *SCROLL* 55
   - 8.1. The Basic Ingredients of the Embedded DSL *SCROLL* 56
   - 8.2. Basic Implementation Concepts 57
   - 8.3. The Usage Layer 61

9. The Metaobject Protocol of *SCROLL* 65
   - 9.1. The Configuration Layer 65
   - 9.2. The Metaobject Protocol Layer 67
   - 9.3. The Specification Layer 85
   - 9.4. Programming the Robotic Co-Worker with *SCROLL* 90

10. Technical Limitations 95
10.1. Limitations and Alternatives 95
10.2. The *SCROLL* Compiler Plugin 97

11. Evaluation 101
11.1. Qualitative Evaluation 101
11.2. Quantitative Evaluation 111
11.3. Discussion 120

12. The Advantages of *SCROLL* 121

IV. Related Work, Conclusion, and Outlook 123

13. Roles with Patterns or other Programming Languages 125
13.1. Roles with Patterns 125
13.2. Roles with other Programming Languages 126

14. Dispatch Models 141
14.1. ALIA4j 143
14.2. Multi-methods: Prototypes with Multiple Dispatch 145
14.3. Korz 146

15. Conclusion 147

16. Future Work 151

V. Appendix 153

A. A Variability Analysis for Roles at Runtime 155

B. An Overview of Scala 173

C. Additional Information 181

D. Source Code 183

List of Figures 261

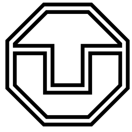
List of Tables 263

List of Listings 265

Index 267

Abbreviations 271

Bibliography 273



TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Software- and Multimedia-Technology

Software Technology Group

PART I.

INTRODUCTION

1

THESIS TOPIC

> *"Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained - on the contrary! - by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously."*
>
> Dijkstra [1974]

In the modern software world, software systems are required to adapt to a changing environment. During the lifetime of a software system, new features are requested, existing requirements change, as well as the underlying hardware and operating systems are regularly being renewed. Software and software libraries once written for a specific purpose may become useful in situations, the developer did not anticipate at the time of their creation. One programming paradigm, object-oriented programming, is widely being used to build extensible and flexible software systems. It was and still is successful, because it supports programming with data structures that closely resemble the problem domain. However, future software systems require a higher level of dynamism, which is not offered by classic object-oriented concepts. Dynamically typed, object-oriented scripting languages (for short, dynamic languages), such as Ruby and Python, have gained popularity not only because of their ease of use, and have created vibrant communities. They enable the extension of modules, classes, and object through concepts such as duck-typing [Python Software Foundation, 2016b]. But programming in a dynamically typed language comes at a cost: without static type information, it is not possible to analyze programs statically and catch many classes of programming errors (e.g., type errors) early on. The burden is solely left to the programmer. For this very reason, Python 3.5 introduced optional support for type annotations via PEP-484 [Python Software Foundation, 2016a] that can be used by tools such as static type checkers and documentation generators.

The influence of roles on language design is in the focus of this work, especially the accompanying problems and ambiguities when dealing with dynamically evolving objects. Role-based programming has been proposed as an extension to object-oriented programming, introducing extensionality in a controlled and well-defined manner. It has been motivated by an easy-to-understand analogy. Also in the real world, objects play different roles in different contexts. In essence, it enables objects to modify and extend their behaviors dynamically at runtime, without the limits imposed by the class hierarchy. The initial idea for role-based programming dates back to 1977 [Bachman and Daya, 1977] and has been discussed in many publications. Many fundamental research topics, such as the definition of requirements for a role-based language [Steimann, 2000a], the

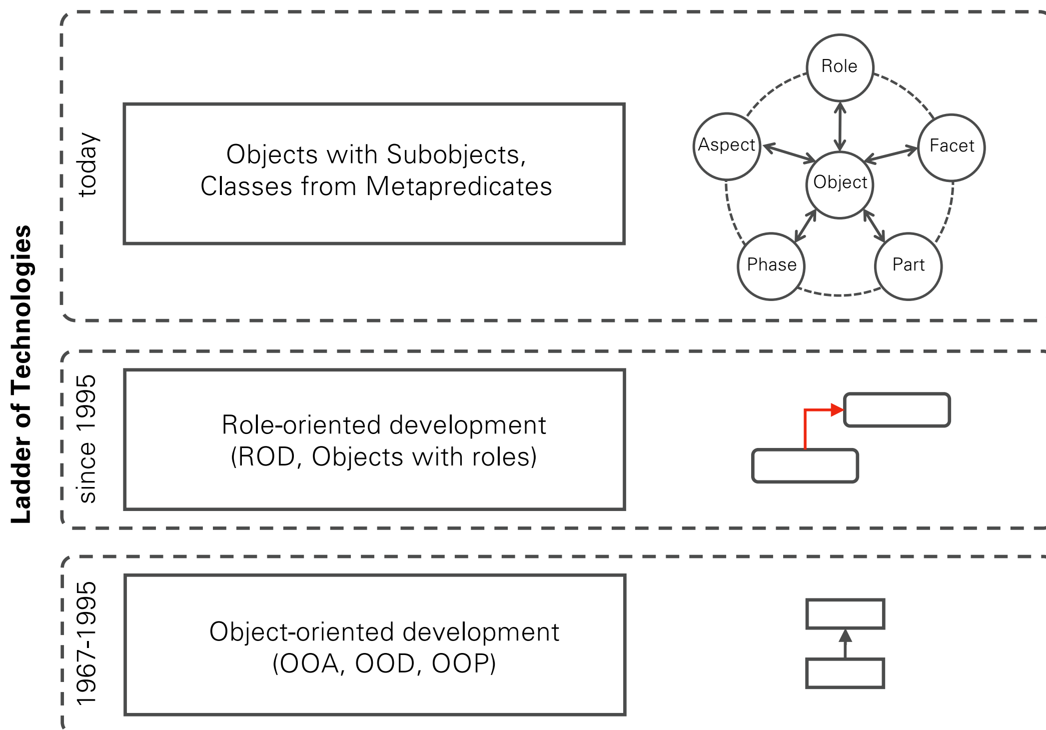


Figure 1.1.: The ladder of technologies from classical object-oriented development up to the current satellite-oriented approach.

definition of a metamodel, and a common terminology [Kühn et al., 2014] have been published. How to simply represent roles in existing language runtime environments remains as an open question. Current implementations rely on proxies, reflection, runtime weaving and runtime code generation to support mechanisms, such as true delegation and dynamic role dispatch [Arnaudo et al., 2007; Herrmann, 2007]. This requires additional management code and leads to more problems, such as incomprehensible error messages with polluted stack traces. Furthermore, existing role-based programming languages only support a small subset of the desired role features. Especially, they lack a well-defined concept for context- and role-aware method dispatch to overcome ambiguities, which are introduced with roles. These points are the major roadblock for the wider adoption of role-based programming. The goal of this thesis is to research how roles can be represented at runtime and supported by a rich dispatching concept. A prototypical implementation, called *SCROLL*, was developed and is introduced by various examples and an in-depth evaluation.

2

CONTRIBUTIONS AND OUTLINE

Scripting languages like Python, JavaScript or Ruby offer flexible object semantics to the developer. On the one hand, programmers can rely on classic object-oriented features, such as inheritance, encapsulation and polymorphism, and on the other hand, they are able to add and remove members from existing objects, as well as merge them at any given point in their life-cycle [Menon et al., 2013]. This functionality is not available in statically typed object-oriented languages. With roles as first-class citizens in programming, one is able to capture the context-dependent and dynamic parts of objects with their specific behavior and structure in separate role types. At runtime, role objects can be added to their player objects. On the downside, implementing such dynamic objects introduces subtle ambiguities, which need to be handled by a context-aware dispatch at runtime. To overcome these problems, this thesis provides the following main contributions:

(1) **_SCROLL_ and the _SCROLL_ MOP** First, the embedded method-call interception Domain-Specific Language (DSL) *SCROLL* and its underlying Metaobject-Protocol (MOP) [Lämmel, 2002; Mernik et al., 2005; Kiczales et al., 1991] are presented. They are implemented in a lightweight library that allows for pure embedding [Hudak, 1998] of roles in a modern, statically typed, object-oriented language (Scala). Only features that are available through Scala's standard compiler are utilized. *SCROLL* allows for easy integration of legacy code and provides a high degree of separation of concerns.

(2) **A coupling of static and dynamic role typing** With the specification of context-dependent behavior and structure in separate role types, static type checking and program analysis is limited. Static typing leads to early detection of programming mistakes through code analysis, better documentation in form of type-signatures, more opportunities for compiler-optimization, higher runtime-efficiency and an improved design-time development experience. Dynamic typing supports easy prototyping, change to unknown requirements or unpredictable data and application integration. Therefore, coupled static and dynamic role typing supports the developer with the best of both worlds. He benefits from the aforementioned advantages of a statically-typed host language, while at the same time, he profits from the flexibility of dynamic objects.

(3) **A simple implementation pattern for roles in structured contexts** The implementation pattern behind *SCROLL*, to implement context-dependent objects with roles of them specified in separate role types, is presented. This pattern requires only three fairly basic components, namely, compiler rewrites, implicit conversions, and a definition table.

(4) **A role-based dispatch at runtime** A declarative and parameterizable approach for four-dimensional dispatch for roles in structured contexts is described. This approach relies on the representation of dispatch rules as function objects [Stroustrup, 1995].

(5) **Strong type-safety for role-based dispatch** The role-based dispatch in *SCROLL* benefits from being embedded in a statically-typed host language. Nevertheless, type checking suffers from restricted type-safety when handling the dynamic parts of role-playing objects. As a solution to this problem, the dynamic type checking during the role-based dispatch is enriched by additional typing information constructed via introspection [Bobrow et al., 1993] while the static type checking is improved by an optional compiler plugin using static program analysis.

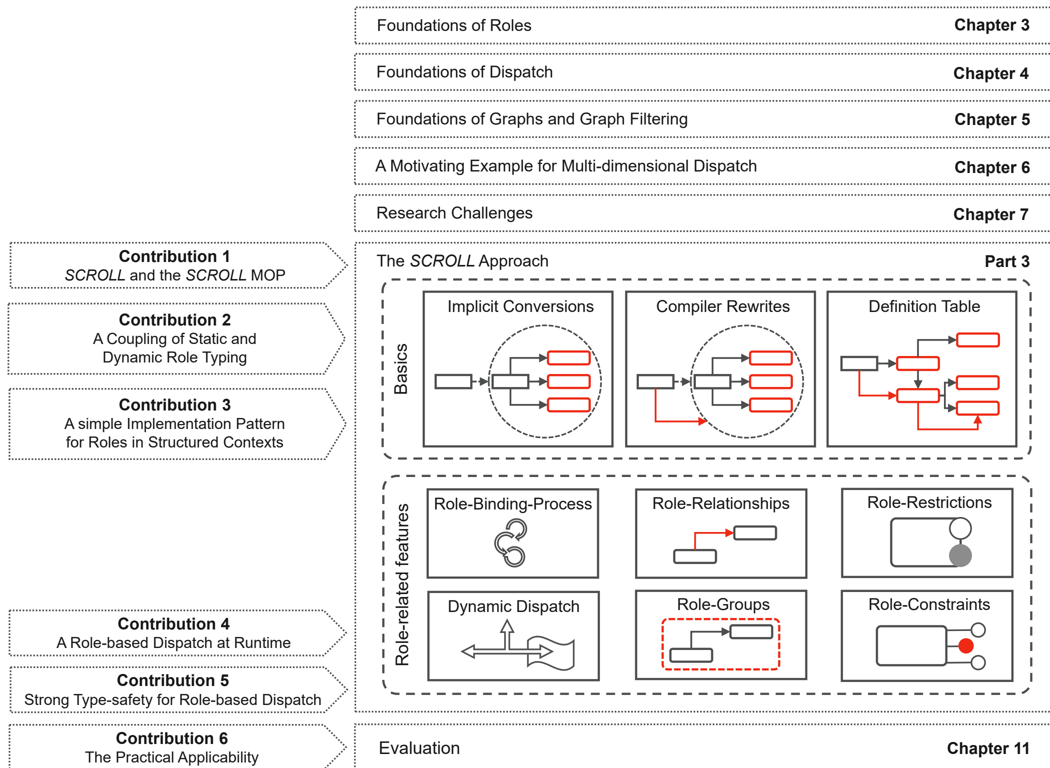
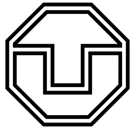


Figure 2.1.: Thesis contributions and outline.

(6) The practical applicability Finally, we show the practical applicability of the proposed approach for the pure embedding of roles by implementing a robotic co-working scenario. It explores the role-based adaptation for the collaboration of humans and robots in a partially unknown environment.

The remainder of this thesis is structured as shown in Fig. 2.1. With the **Background and Problem Analysis** part, the foundations of roles (Chapter 3) and dispatch (Chapter 4) are introduced. These chapters provide an overview of their respective modeling primitives and various notions. A classification of different role notions is presented on the basis of two surveys (Kühn et al. [2014]; Steimann [2000b]). An example implementation with the pure embedding of roles for a robotic co-working scenario is shown in Chapter 6. In this example, role-based adaptation for the collaboration of humans and robots in a partially unknown environment is demonstrated. With that, issues of the classic object-oriented design are discussed (Sect. 6) and requirements for this work are derived (Chapter 7). Based on these foundations and requirements, in the part **Roles in Structured Contexts at Runtime**, an introduction and overview of *SCROLL* is presented (Chapter 8). Its basic ingredients are explained in more depth (Sect. 8.2). The approach for four-dimensional, context-aware dispatch at runtime in the context of roles is described in Sect. 9.3. Furthermore, limitations and technical alternatives are discussed (Chapter 10.1). Afterwards, *SCROLL* is evaluated qualitatively (Sect. 11.1), as well as quantitatively using performance benchmarking (Sect. 11.2). At the end, the *SCROLL* approach is put into the context of available **Related Work** from the contemporary literature (Part IV). The last two chapters give final remarks (**Conclusion**, Chapter 15) and a perspective on future works (Chapter 16).



TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Software- and Multimedia-Technology

Software Technology Group

PART II.

BACKGROUND AND PROBLEM ANALYSIS

3

FOUNDATIONS OF ROLES

Dynamic and adaptive infrastructures are the cornerstone of today's software development, e.g., in the Web 3.0 - the internet of things. In most classic class-based object-oriented systems, the association between instances of a class and the class itself is permanent [Gottlob et al., 1996]. Such systems hardly cope with new requirements during runtime. Class hierarchies need to be carefully planned and laid-out for dynamic extensions. Indeed, they grow exponentially in case the objects they describe are changing [Fowler, 1997].

Two main principles of abstraction are known to structure object-oriented programs: *classification* and *generalization*. The first describes the principle to group objects to classes sharing behavior and attributes. The second principle means the organization of those classes into class hierarchies with grouping the common behavior and attributes into a new superclass, which is then shared among all subclasses. However, when describing real-world objects embedded in a fast and frequently changing environment as well as their classification in a class hierarchy, the permanent association between instances and their classes appears to be too inflexible, because it cannot cope with the requirements of those fast growing or changing systems. When extending an object, the object must be replaced. Then, the internal state of objects needs to be copied, which renders their management cumbersome and error-prone. It becomes even more difficult if there are several of those changes at the same time for the same object. Introducing separate classes for each combination of new behavior for adaptation leads to fast growing class hierarchies, which is undesired.

classification
generalization

The concept of roles was introduced by Bachman and Daya [1977] as an extension to the network data model. It enables the addition and removal of behavior and attributes at runtime to objects providing a substantial advantage over traditional programming languages, such as Java. Over the recent decades, a lot of role-based approaches have been proposed in the literature, all providing a different notion of roles. This section starts with a brief historical introduction and focuses on the features and basic notions of roles, later on. As a sophisticated role notion as base for this thesis, the Compartment Role Object Model (CROM) [Kühn et al., 2014] is discussed.

3.1. THE HISTORICAL TERM ROLE

Different definitions can be found in the contemporary literature with regard to the term roles. Being the first one introducing this concept, Bachman and Daya [1977] provides a vivid description with an example explaining the differences between the role concept and the classic theory of entities in databases:

> *"The use of the word role is taken from the theatrical context where a role is defined to be a part played by an actor on a stage. It is a defined behavior pattern which may be assumed by entities of different kinds. Furthermore, a particular entity may concurrently play one or more roles. Hence, the existence of all the roles of interest for a given entity characterize that entity. It is the author's contention that most conventional file records and relational file n-tupel are role oriented. These files typically deal with employees, customers, patients, or students, all of which are role types. This role orientation is in contrast with integrated database theory which has taught that each record should represent all aspects of some one entity in the real-world. This difference*

> *in viewpoint has caused a great deal of confusion. The reason for the confusion is understood when it is realized that neither the roles of the real-world nor the entities of the real-world are subset of the other."*
> [Bachman and Daya, 1977, p. 465]

This passage summarizes the basics of the role concept. It offers an intuitive way to model real-world objects. Today, many systems incorporate the concept of roles. Database systems, implementations for security and access control, artificial intelligence, or agent-based systems may be referenced here. Nevertheless, first-class support in widely used general purpose languages, e.g., Java or C++, is still missing. There are many definitions of the role metaphor floating around in the research landscape, focusing on different aspects tailored to various use-cases. As an example, Graversen [2006] defines roles as follows:

> *"A role is a specification of properties (methods and states). The methods are composed with the intention of making an object function in a specific context, thus all the properties of the object are also available to the role. For example, when a person plays the role of a professor, the professor still has a name (the name of the person). A key feature for roles is their ability to be attached at runtime and on object basis. This stands in contrast to normal inheritance that operates on types and is applied at compile-time. Conceptually, roles are multiple disjoint classifications of an object. Another key feature of roles is the property of multiple views: A role attached to an object does not invalidate the properties of the object."*
> [Graversen, 2006, p. 28]

On the other hand, with respect to role-based modeling in agent-based systems, Kendall [1999] describes roles as:

> *"A role model identifies and describes an archetypal or recurring structure of interacting entities in terms of roles. A role defines a position and a set of responsibilities within a role model. A role has collaborators; these are other roles that it interacts with. A role's services and activities are accessible through an external interface; usually there is a distinct interface for each collaboration path between two interacting roles."*
> [Kendall, 1999, p. 1]

What are the required definitions and descriptions for the concept of roles to incorporate the static and dynamic aspects of problems in their respective domains? This will be investigated in the following sections.

3.2. ISSUES OF OBJECT-ORIENTED SOFTWARE DESIGN

Each object-oriented software design has to solve the static and dynamic aspects of the following problems [Riehle, 2000]. The interactions of a class or its instances with the client are non-trivial (*class complexity* **/P.1/**). Each of those clients handle the instances differently and with different use-cases in mind. This interaction needs to be described and constrained from the viewpoint of all relevant contexts. Furthermore, a sound understanding of *object collaborations* and relations at runtime is crucial (**/P.2/**). Adaptive software systems need to be able to define and check those collaborations. Once they are understood, it is important to group and separate them for the sake of comprehensibility

class complexity

object collaborations

(*separation of concerns* **/P.3/**). Relations need to be assigned to relevant contexts. Derived from the two aforementioned tasks, *reuse* is one of the most important attributes, on which adaptivity of the resulting system benefits from (**/P.4/**). With the *invariants and constraints*, behavior of collaborating objects can be constrained and checked during runtime (**/P.5/**). Finally, in adaptive software systems, it is impossible to foresee each and every possible future use-case and context. Hence, all problems, listed so far, define basic requirements that allow for adapting to new and unforeseen contexts (**/P.6/**). Role modeling delivers solutions for these problems:

separation of concerns

reuse

invariants and constraints

Class complexity (/P.1/) Adding a set of role types to a class, thus, splitting context-sensitive and context-free behavior, reduces the class complexity. A role type exactly specifies how the instance of a class interacts in a certain context.

Object collaborations (/P.2/) and separation of concerns (/P.3/) The major part of the complexity of adaptive software infrastructures stems from complex object collaborations. They become manageable with the specification of smaller, hierarchically decomposable role models. Each role model describes an individual aspect of the object collaboration.

Reuse (/P.4/) With the clear separation of object collaborations into smaller pieces (i.e., role models) with regard to the concerns of the problem space and their composability, a high amount of reuse is enabled.

Invariants and constraints (/P.5/) A role type is a good target for invariants and constraints.

Adaptivity (/P.6/) Adding role types dynamically allows for adapting unforeseen contexts.

The following section describes the motivation for roles in general and introduces their major ingredients in more detail.

3.3. THE INTRODUCTION OF ROLES

Abstracting and simplifying the representation of the real-world for some given use-case is, in general, called *modeling*. A model omits irrelevant parts of the captured world. An *object-oriented model* abstracts objects to classes and types. Hence, the complex relationships of real-world objects are simplified to the pertinent needs for the desired purpose. The result of the modeling process is then used during the analysis or design phase in software development. Here, two major goals can be targeted. First, capturing real situations of a domain for communication during the analysis and, secondly, representing the design of the software system itself, describing its elements and interactions [Steimann, 2000b]. Traditional modeling languages, such as the Unified Modeling Language (UML) or Entity-Relationship Model (ER), only consider entities of fixed types (static typing), pre-defined structure (attributes), and behavior (methods) without being able to change them dynamically. Every aspect of an entity needs to be integrated at design time, even though some attributes or functions are not needed in certain contexts. Often, the adaptation is handled by static inheritance. This is problematic since, e.g., subtyping leads to an enormous amount of subtypes for each and every new context the entity is intended to interact in (*combinatorial explosion of subtypes*). Extensibility becomes, consequently, impracticable. Finally, one might not be able to decide whether a specialization or generalization is applicable. Using design patterns can solve certain problems, but introduces

modeling

object-oriented model

combinatorial explosion of subtypes

additional overhead with regard to maintenance and readability. This underlines the need for more flexibility and dynamics to improve modeling.

In classic programming languages, the situation is quite similar. During its lifetime, an object might need to change its behavior and attributes for a certain period. In addition, entities of the same type may have different attributes and behavior at the same time. Those aspects are usually not directly supported within classical object-oriented programming languages with static typing. An entity is part of a certain type throughout its lifetime. When recreating it with a different type at runtime, it loses its identity, state needs to be copied, and clients have to handle the newly created identity causing the need for additional management code. An additional problem occurs when an object is conceptually separated into multiple individual sub-objects (*split-object problem* [Dony et al., 1992]). This requires even more additional management code complicating code maintenance. The split-object problem is very much similar to the issue of *object schizophrenia*: *"Object schizophrenia results when the state and/or behavior of what is intended to appear as a single object are actually broken into several objects (each of which has its own object identity)."* [Harrison, 2016].

split-object problem

object schizophrenia

In the end, with modern software systems becoming more and more complex and having to adapt to continuously changing environments, the resulting problems during design- and runtime cannot be managed easily anymore. Approaches, such as dynamic aspect-orientation [Popovici et al., 2002] and context-oriented programming [Hirschfeld et al., 2008] have been introduced by researchers in the past to handle the aforementioned problems of too static software systems. As an alternative, role-oriented programming [Steimann, 2000a,b; Boella et al., 2007; Graversen, 2006; Gottlob et al., 1996; Pernici, 1990] can be employed, which is in the focus of this thesis. With the idea of separation of concerns in mind, dynamic and flexible parts of an object are modeled separately from the entity's core. Several parts of an entity have different lifetimes and may only exist during a certain period of the entity's lifetime. These entities are split into natural types (entity's core type) and role types. This enables context-dependent structural and behavioral adaptation. Thus, the dynamic evolution of entities over time becomes an integral part of modeling and programming. Those evolving objects are explicitly modeled and represented at runtime accordingly. The role as a concept, modeling primitive, and first-class citizen in programming captures the context-dependence and dynamic parts of objects with their specific behavior and structure in separate types. Hence, role-based type systems explicitly model behavioral adaptation, in contrast to traditional static type-systems. Entities can evolve during runtime without changing their natural type at all. Instead, they start and stop playing roles. This enables modeling and implementing complex, context-dependent behavior of objects in frequently changing environments. We conclude this introduction to the role concept with a list of arguments in favor of the role concept, which also acts as a motivation for this research effort:

- Roles are an easy-to-understand concept already known from the real-world.
- Since roles capture context-dependent behavior, they introduce a controlled form of dynamic adaptation which is easy to reason about, since this adaptation is always scoped to certain contexts.
- Many problem domains can take advantage of a design and implementation based on roles. In many cases, this leads to a software design that is easier to understand for humans [Steimann, 2000a].
- An embedding of roles in object-oriented programming is missing. The Role Object Pattern [Bäumer et al., 1997] has been developed to emulate role features in

classic object-oriented languages. This indicates the need for native role support in programming languages.

- Finally, roles allow for easier reuse, since the usual restrictions of strict object-oriented class hierarchies do not apply. Roles permit arbitrary, fine-grained extensions to an object's behavior. If the role concept is applied consistently, the natural types contain only a minimal set of necessary fields and methods reducing the risk of conflicts during reuse. Cross-cutting features can be extracted into reusable role types.

In summary, the role concept is essentially an extension to object orientation enabling objects to adapt their behaviors dynamically. At runtime, roles are bound to objects which then become role-players.

With the extension of object orientation, this thesis assumes a basic knowledge of the terminology used in research on object orientation. Nevertheless, delegation, forwarding, and consultation are shortly explained in the following. For that, we define two terms beforehand. A *sender* is the object that sends a message to another object (the receiver). Thus, the *receiver* is the object that receives a message from the sender. The term *delegation* denotes that when a message cannot be understood by a receiver, the message is passed on to another object. Here, no assumption is made whether the sender and the receiver are completely separated or the sender object aggregates the receiver object. The difference between *forwarding* and delegation is that in forwarding the receiver is definitely a separate object. Finally, *consultation* denotes that when a message cannot be understood by a receiver, the message is passed on to an object aggregated in the sender object. Furthermore, a *compound object type* is a subclass of a conglomerate of the player type and its role types. An instance of such a compound object type is called *compound object*. The type of the compound object built from a player of type `A` and roles `R1` and `R2` is the tupled type `{A, R1, R2}`. Consultation is used for the communication between the aggregated parts of compound objects.

sender

receiver

delegation

forwarding

consultation

compound object type

compound object

3.4. TOWARDS A COMMON UNDERSTANDING OF ROLES

In our life, communications, speech, and in the way we think, the term role and its concepts are well-known. In various research areas (e.g., linguistics and sociology) this term is widely used and accepted. Going back to the origins of the concept, in a theater, a role refers to a figure an actor plays. It is something abstract to describe how someone can act. A role is defined completely independent of its player and gets filled with life by the actor (player) adopting its behavior and structure. Similar ideas hold in software modeling and programming as well. We use roles to dynamically add and remove behavior and structure during the runtime and lifetime of an object.

Each approach on role-oriented programming that appeared in the literature during the last decades utilizes its own interpretation of roles leading to an inhomogeneous research landscape. Hence, there is no common notion of what a role is. For instance, ER or UML only consider roles as named places in associations without taking attributes or behavior into account. On the other hand, programming languages such as Object-Teams/Java (OT/J) [Herrmann, 2002] allow for dynamically adapting the behavior of objects at runtime. To provide a universal formal role modeling language, Steimann surveyed approaches until the year 2000 [Steimann, 2000a,b]. Based on this, 15 different questions on the term role were identified. Some questions are conflicting; it seems that no role-based programming language will ever be able to realize all of those features. Another survey analyzed role-based modeling and programming approaches between

1. Roles have properties and behaviors.
2. Roles depend on relationships.
3. Objects may play different roles simultaneously.
4. Objects may play the same role (type) several times.
5. Objects may acquire and abandon roles dynamically.
6. The sequence of role acquisition/removal may be restricted.
7. Unrelated objects can play the same role.
8. Roles can play roles.
9. Roles can be transferred between objects.
10. The state of an object can be role-specific.
11. Features of an object can be role-specific.
12. Roles restrict access.
13. Different roles may share structure and behavior.
14. An object and its roles share identity.
15. An object and its roles have different identities.
16. Relationships between roles can be constrained.
17. There may be constraints between relationships.
18. Roles can be grouped and constrained together.
19. Roles depend on compartments.
20. Compartments have properties and behaviors.
21. A role can be part of several compartments.
22. Compartments may play roles like objects.
23. Compartments may play roles which are part of themselves.
24. Compartments can contain other compartments.
25. Different compartments may share structure and behavior.
26. Compartments have their own identity.

Figure 3.1.: Steimann's 15 classifying features (1-15), extracted from [Steimann, 2000b] and the additional ones (16-26) with regard to the context-dependent nature of roles [Kühn et al., 2014].

the years 2000 and 2014 [Kühn et al., 2014]. An overview of the identified questions of Steimann and Kühn et al. is given in Table 3.1. The authors in Kühn et al. [2014] focus more on the three different aspects roles try to serve, namely their *relational*, *contextual*, and *behavioral* nature. The relational aspect denotes that different entities interact with each other, or are connected by using roles. With the contextual aspect, roles may be utilized to describe context-dependent features of entities. Finally, roles address behavioral aspects of entities, which refer to the dynamic set of attributes as well as to methods. Based on these three perspectives, one can derive the following classes of approaches of role-based languages:

Relational nature Languages such as UML or Rumer [Balzer et al., 2007] only focus on the relational aspects of roles. They are used to connect objects by roles as named places in relations (UML within associations, Rumer with relationships).

Behavioral nature Chameleon [Østerbye, 2003] and Rava [He et al., 2006] focus on adaptation of behavior and structure only. Adapting behavior and structure with regard to certain situations is in the focus of contextual role languages such as, OT/J [Herrmann, 2005] or NextEJ [Kamina and Tamai, 2009].

Contextual nature There are currently no known languages solely focusing on the context-dependent nature of roles.

Combining relational and contextual nature However, the HELENA approach [Hennicker and Klarl, 2014] combines the relational and context-dependent interpretation. Roles are used to declare an objects' interaction with other objects embedded in a context. Sadly, behavioral and structural adaptation are not considered in this class of approaches.

Combining relational and behavioral nature Here, relational as well as behavioral features are brought together (e.g., in ORM [Halpin, 2006], Lodwick [Steimann, 2000a] or the INM [Liu and Hu, 2009]).

Combining all three natures Finally, the last category combines all three aspects of roles. CROM is the only proposed metamodel known so far falling into this category [Kühn et al., 2014]. Hence, it is chosen to form the conceptual basis of the implementation of *SCROLL*.

Following that classification, it is necessary to further investigate the heavily overloaded term *context*. Context can be understood as environmental information and as objectified collaboration containing other entities. The first interpretation can be described as follows:

context

> *"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*
> [Abowd et al., 1999, pp. 306–307]

A context surrounds an entity and provides additional information. Information may be time, place, temperature, or the state of the application running. Paradoxically, even the lack of information about an entity can be regarded as an information. Furthermore, entities are always attached to a specific context. For instance, sensor data (like GPS) is only valid for a certain device and its user. For other users not currently present, this information may not be relevant. In sum, this general context definition describes an entity's environmental information that has no specific identity, no intrinsic behavior and is omnipresent. In contrast, a *compartment* as introduced by Kühn et al. [2014] is defined as:

compartment

> *"[W]ithin modeling languages, context represents a collaboration or container of a fixed, limited scope. To overcome this dichotomy, researchers avoided the term context by using other terms, i.e., environments, institutions, teams, and ensembles. In turn, we use the term compartment as a generalization of these terms to denote an objectified collaboration with a limited number of participating roles and a fixed scope."*
> [Kühn et al., 2014, p. 146]

While a context (e.g., a cold and rainy day in London) is intentional (described by rules or attributes), without its own identity, intrinsic behavior or existential parts and with an indefinite lifetime - a compartment (e.g., a first-class train car) is extensional, i.e., is explicitly specified. Its instances carry identity, have behavior, state, a defined lifetime and contain roles as its parts. Hence, a compartment can be seen as an objectified collaboration for the contained roles. This thesis is based on this definition of compartments, as introduced by CROM, which is discussed in the following.

Concept	Rigidity	Foundedness	Identity	Example
Natural Types	rigid	non-founded	unique	Person
Compartment Types	rigid	founded	unique	University
Role Types	anti-rigid	founded	derived	Student
Relationship Types	rigid	founded	composed	takes class

Table 3.1.: Ontological foundation of meta types within the Compartment Role Object Model [Kühn et al., 2015].

3.5. THE COMPARTMENT ROLE OBJECT MODEL (CROM)

CROM is a metamodel for constructing role-based models with an additional Compartment Role Object Instance (CROI) for the representation at instance level, and a Constraint Model for specifying various constraints for additional validation.

natural relationship types rigidity foundedness identity

In essence, four different metatypes are defined in CROM: *natural*, *compartment*, *role*, and *relationship types*. As a formal basis, the ontological properties *rigidity*, *foundedness*, and *identity* are applied [Albuquerque and Guizzardi, 2013]. The first concept denotes that an instance has to be a member of this type for its whole lifetime. The opposite is an anti-rigid type. Foundedness means that a type is existentially dependent on the existence of other types. Instances cannot exist on their own, but only in collaboration with partners. Finally, *identity* characterizes whether a type's instance has a unique, derived, or composed identity.

Based on that, distinguishing the aforementioned four metaclasses of CROM is now possible (Table 3.1). Natural types (*NT*) are considered to be rigid, non-founded, and have a unique identity. Compartment types (*CT*) are rigid, but founded and have a unique identity. Role types (*RT*), as the opposite of natural types, are anti-rigid and founded, while their identity is derived from their player types. Finally, relationship types (*RST*) are rigid, founded and have a composed identity. The type of a relationship cannot be changed and requires at least two role types to participate in. The identity of a relationship is composed by the participating roles. All the four metatypes do not live in isolation. They are connected by relations and functions with special semantics, as shown in the following:

fills relation This relation binds player types with role types. Hence, it defines that objects of a certain player type can only play roles of a certain role type. The player can be a natural type or a compartment type. The fills relation is not limited to only one player type per role type. Multiple player types for a single role type are allowed.

parts function The parts function maps each role type to a specific compartment type. Hence, each role type is contained in exactly one compartment type. To guarantee the foundedness property of compartment types, each compartment must at least contain one role type.

rel function This function maps relationship types to distinct role types in the same compartment type. Relationships spanning multiple compartments are not allowed within CROM. This is considered to be ill-modeled by the authors [Kühn et al., 2014], as related roles in different collaborations or situations should obviously belong to the same compartment.

CROM is defined as a tuple over these concepts: $M = (NT, RT, CT, RST, \text{fills}, \text{parts}, \text{rel})$. In addition, the instance representation CROI was proposed. On this level, natural,

Notation	Meaning
$\mathbb{N}_1, \mathbb{N}_0$	Natural numbers excluding 0; Natural numbers including 0
$<, \leq, >, \geq$	Lesser; Lesser than; Greater; Greater than
$\wedge, \vee, \neg, \Rightarrow$	Predicate logical and, or, not, implication
$\exists, \forall$	Predicate logical existential- and all-quantor
$=, \Leftrightarrow$	Equality; Predicate logical biconditional
Crossed out	Relation negation (e.g., $a \neq b$ means $\neg(a = b)$, $a \not\subset S$ means $\neg(a \subset S)$, $v \not\sim p$ means $\neg(v \sim p)$, etc.)
$(a, b, \ldots)$	Tuple with first element a, second b, etc.
$\text{proj} : \mathbb{N}_1 \times \top \to \top$	Tuple element projection (e.g., $\text{proj}(2, (a, b, c)) = b$)
$\text{size} : \top \to \mathbb{N}_0$	Tuple size (e.g., $\text{size}((a, b, c)) = 3$)
$\{a, b, \ldots\}, \emptyset$	Set containing elements a, b, etc.; Empty set
$\{e \mid c\}$	Set comprehension (set containing all elements e for which c holds)
$\cup, \cap, \setminus, \in, \|S\|$	Set union, intersection, difference, membership and cardinality
$\subset, \subseteq, \supset, \supseteq$	Proper subset, subset, proper superset and superset relation
$\times$	Cartesian product
$\top$	Any valid element (e.g., let $A = \{(2, a), (1, b), (1, c)\}$ then $(\top, d) \notin A$ but $(1, \top) \in A$ whereas $\top = b \vee \top = c$)

Table 3.2.: Notations overview.

compartment, and role types are instantiated to naturals (N), compartments (C), and roles (R). Again, they are connected and related with the following relations and functions:

plays relation This relation connects player, role and compartment instances. Here, only a single player per role is allowed. While *fills* represents a can-play semantic on type level, on the instances, *play* denotes the actual playing of a certain role.

fills

play

type function Each instance has its type. Thus, this function returns the type for a given instance.

links function This function keeps track of the information about roles participating in relationships. It returns related roles for a given relationship type.

CROI is finally defined as the tuple: $i = (N, R, C, \text{type}, \text{plays}, \text{links})$ [Kühn et al., 2014]. Additionally, a Constraint Model is presented. It constrains roles and relationships (e.g., to be irreflexive) and introduces role groups [Kühn et al., 2015]. Those groups feature role group constraints, e.g., cardinality constraints. They are applied to the whole group instead of each individual contained role. A constraint can be, for instance, one of the following already known from pre-existing research [Riehle, 2000]. Let RT be the set of all role types in a given role model. For each pair (A, B) from role types A and B out of RT one of these constraints can be defined:

role-implied constraint An object playing a role of role type A is required to also play a role of role type B (but not necessarily the other way round).

role-equivalent constraint An object playing a role of role type A is required to also play a role of role type B, and vice-versa.

role-prohibited constraint An object playing a role of role type A is never allowed to play a role of role type B, and vice-versa.

role-dontcare constraint No constraint is applied for an object playing a role defined in the pair (A, B).

role group

With a *role group*, those constraints can now be applied to a group of role types via cardinality constraints instead of denoting them explicitly to each and every type. For instance, instead of applying a prohibition constraint on each role type, which can be very cumbersome due to the quadratic growth, the grouping enables the developer to apply a cardinality constraint (that at most n of the corresponding role types can be played) on the whole group. The syntax of role groups is defined as [Kühn et al., 2015]:

Definition 1 (Syntax of Role Groups)
Let RT be the set of role types; then the set of role groups RG is defined inductively:

- If $\mathrm{rt} \in RT$, then $\mathrm{rt} \in RG$, and
- If $B \subseteq RG$ and $n, m \in \mathbb{N} \cup \{\infty\}$ with $n \leq m$, then $(B, n, m) \in RG$.

Definition 2 (Atoms Function)
Let $\mathcal{M} = (NT, RT, CT, RST, \mathrm{fills}, \mathrm{parts}, \mathrm{rel})$ be a well-formed CROM; then $\mathrm{atoms} : RG \rightarrow 2^{RT}$ is a function, defined as:

$$\mathrm{atoms}(a) := \begin{cases} \{a\} & \textbf{if } a \in RT \\ \bigcup_{b \in B} \mathrm{atoms}(b) & \textbf{if } a \equiv (B, n, m) \end{cases}$$

The *atoms* $: RG \rightarrow 2^{RT}$ function recursively collects all role types within a given role group and the *plays* $\subseteq (N \cup C) \times C \times R$ function identifies the objects (either natural or compartment) playing a certain role in a specific compartment. This leads to the following semantics for role groups [Kühn et al., 2015]:

Definition 3 (Semantics of Role Groups)
Let RT be the set of role types of a well-formed CROM $\mathcal{M}$, $\imath = (N, R, C, \mathrm{type}, \mathrm{plays}, \mathrm{links})$ a CROI compliant to $\mathcal{M}$, $c \in C$ a compartment, and $o \in O^c$ an object playing a role in c. Then, the semantics of role groups is defined by the evaluation function $(\cdot)^{\mathcal{I}_o^c} : RG \rightarrow \{0, 1\}$ as:

$$a^{\mathcal{I}_o^c} := \begin{cases} 1 & \textbf{if } a \in RT \wedge \exists r, (o, c, r) \in \mathrm{plays} : \mathrm{type}(r) = a \\ & \textbf{or} \quad a \equiv (B, n, m) \wedge n \leq \sum_{b \in B} b^{\mathcal{I}_o^c} \leq m \\ 0 & \textbf{otherwise} \end{cases}$$

Role groups constrain the set of roles an object o is allowed to play simultaneously in a certain compartment c. In case a is a role type, $\mathrm{rt}^{\mathcal{I}_o^c}$ checks whether o plays a role of type rt in c. If a is a role group (B, n, m), it checks whether the sum of the evaluations for all $b \in B$ is between n and m. In general, all of the role constraints presented above (role-dontcare, role-implied, role-equivalent, and role-prohibited) from Riehle [2000] and role groups correspond to propositional formulas over role types. For instance, a role-implied from consultant to customer would be modeled as: $(\{(\{Consultant\}, 0, 0), Customer\}, 1, 2)$. This is equivalent to the formula $\neg Consultant \vee Customer$ and fulfills the intended semantics of the role-implied. In summary, CROM and CROI build a powerful and, most importantly, formally defined base to build on. Thus, it is used as underlying conceptual motivation for *SCROLL*.

3.6. THE AUTOMATON-BASED ROLE-BINDING PROCESS

The process of binding roles to objects is called *role-binding*. It consists of the binding technique and the binding operation. The binding technique specifies the implementation of the relationships. For that, different role-programming implementations use different design patterns. These design patterns are themselves not based on roles, but on classic object-oriented concepts, and have to solve the problem that references are not bidirectional. Thus, the roles and objects do not just have to record their context, but the context also has to record all collaborations taking place in it. The binding process contains a set of binding operations that are executed in sequence. This process binds the object with the role together in a specific context.

role-binding

In order to play a role (*role-playing*), it first has to be bound (role-binding). Thus, role-binding is the requirement for role-playing. For the implementation of binding roles, an automaton-based approach is used, as discussed in Schütze [2016]. With that, the role-binding is externalized for better separation of concerns. The automaton models the reasons which lead to the binding of a specific role. A *role-playing automaton* is an event-triggered Finite State Machine (FSM) that models the circumstances leading to adding, removing, or transferring specific roles. A FSM can be described as a set of relations of the form [Ericsson Utvecklings AB, 1999]:

role-playing

role-playing automaton

$$\text{State}(S) \times \text{Event}(E) \rightarrow \text{Actions}(A),\ \text{State}(S')$$

These relations are interpreted as follows: if the system is in state S and the event E occurs, the actions A should be performed and a transition to the state S' should be made. Edges in the automaton define reasons and requirements to further adapt the role-playing objects, while the nodes define which roles are bound and which ones are unbound. In sum, a role-playing automaton specifies the role life-cycle with the binding, unbinding or transfer of role instances between objects and is defined as follows:

Definition 4 (Role-playing Automaton)
A role-playing automaton RPA is an event-triggered FSM and is defined as a tuple $(\mathscr{S}, \mathscr{E}, \mathscr{A}, m, \mathscr{P}, \mathscr{R})$ with:

- $\mathscr{S}$ is the set of states in the system,
- $\mathscr{E}$ is the set of events the system reacts on,
- $\mathscr{A}$ is the set of actions the system should perform with $\mathscr{A} :=$ {bind r to p, remove r from p, transfer r from p_1 to p_2}, where $r \in \mathscr{R}$, and $p, p_1, p_2 \in \mathscr{P}$,
- $m : (\mathscr{S}_a, \mathscr{S}_b) \rightarrow (\mathscr{E}_i, \mathscr{A}_j)$ is a function, mapping the transitions between two states $\mathscr{S}_a, \mathscr{S}_b \in \mathscr{S}$ to their respective sets of events (with $\mathscr{E}_i \subset \mathscr{E}$) and actions (with $\mathscr{A}_j \subset \mathscr{A}$),
- $\mathscr{P}$ is the set of affected players, and
- $\mathscr{R}$ is the set of relevant roles, i.e., roles that should be added, removed, or transferred.

With this definition, we construct the example shown in Fig. 3.2. The initial system state should be modified according to the provided role-playing automaton $\text{RPA}_{\text{Example}}$, i.e., the role instances `a` and `b` should be bound to the player instance `o` once the events `EVENT_A` and `EVENT_B` happen. Hence, the role-playing automaton $\text{RPA}_{\text{Example}}$ can be defined as:

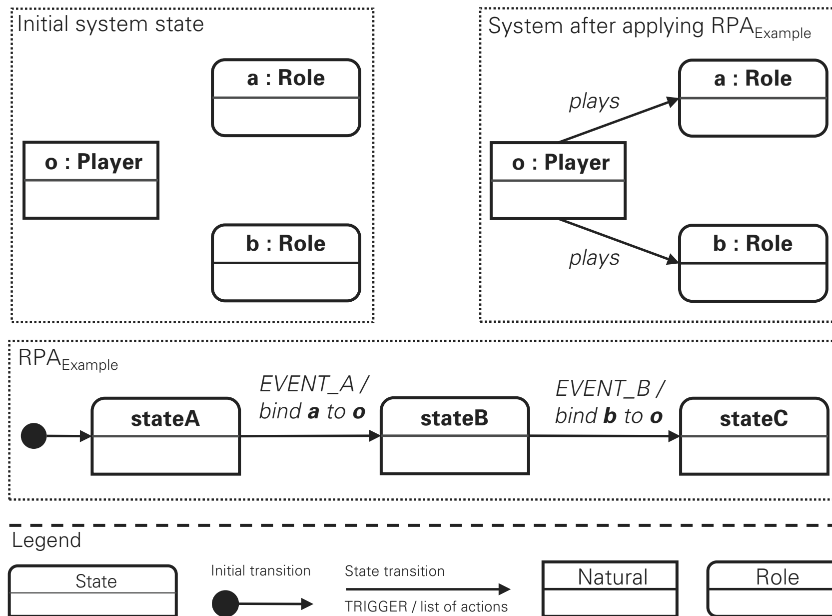


Figure 3.2.: An example for an event-triggered role-playing automaton.

Definition 5 (The Example Role-playing Automaton $\text{RPA}_{\text{Example}}$)
The example role-playing automaton $\text{RPA}_{\text{Example}}$ is an event-triggered FSM and is defined as a tuple $(\mathscr{S}, \mathscr{E}, \mathscr{A}, m, \mathscr{P}, \mathscr{R})$ with:

- $\mathscr{S}$:= {stateA, stateB, stateC},
- $\mathscr{E}$:= {EVENT_A, EVENT_B},
- $\mathscr{A}$:= {bind a to o, bind b to o},
- m(stateA, stateB) = ({EVENT_A}, {bind a to o}),
- m(stateB, stateC) = ({EVENT_B}, {bind b to o}),
- $\mathscr{P}$:= {o}, and
- $\mathscr{R}$:= {a, b}.

3.7. RUNTIME-SPECIFIC ROLE FEATURES

Contemporary literature has not been able to provide a unique definition of what a role is, especially with regard to runtime. These aspects of the semantics of the role concept have been described in a variability analysis [Graversen, 2006]. This analysis relies on the encountered semantics of roles which goes far beyond the analysis presented in Kühn et al. [2014], because many runtime features are investigated. Appendix A presents this with the help of feature diagrams. This list of features for roles at runtime later in this section permits us to reflect over the set of features *SCROLL* covers, which is also used during the evaluation (Sect. 11.1.3).

In the literature, most papers focus on presenting a specific role model. Only very few authors have attempted to compare different role semantics (Kappel et al. [1998]; Kniesel [1996]; Kristensen [1996]; Steimann [2000b]; Kühn et al. [2014]). The papers represent different strategies of role surveys. Steimann's enumerated list (Fig. 3.1) of role features [Steimann, 2000b] is a good initial approach to the understanding of role models. However, the list seems like a long stream of questions, lacking structure. Little evidence

is provided whether a topic has been fully covered. Furthermore, the list approach lacks the ability to show legal and illegal combinations of the details of role semantics and mixes runtime and conceptual level.

Several authors have attempted to introduce more structure into the analysis of features of roles. The two-dimensional schema presented in Kappel et al. [1998] remedies some problems associated with Steimann's list approach. In effect, the first column of the schema is a list of criteria. Each requirement is evaluated by answering supported, not supported and unknown, respectively. This makes it much clearer which languages support which feature of roles. Although the domain of Kniesel's analysis [Kniesel, 1996] is very limited, his strategy enables him to fully explore each dimension. More importantly, using symmetry considerations, the comparison enables the synthesis of unexplored role semantics. The drawbacks are that the comparison is difficult to read and comprehend upon first contact. Additionally, it is problematic for the reader to simultaneously focus on many dimensions of semantics, which is required by the structure of the model. Finally, the survey presented in Kühn et al. [2014] (that is also discussed during the evaluation of *SCROLL* in Sect. 11.1.2) builds on top of Steimann's role features and presents 11 more features tailored to the relational and contextual nature of roles (Fig. 3.1). Nevertheless, it suffers from the same problems as Steimann's survey and does not make a clear distinction between the instance (runtime) and conceptual level.

Another, well-structured approach to describe the semantic variability for roles at runtime has been presented in Graversen [2006] with the help of feature diagrams as discussed in Appendix A. To be able to compare this set of features with the question lists of Steimann [2000b] and Kühn et al. [2014] for the evaluation of *SCROLL* (Sect. 11.1.3), we derived a list of features for roles at runtime (see Table 3.3), extending the list presented in Fig. 3.1. Some overlapping features were omitted (e.g., identity-related features). Graversen [2006] introduces some terminology, which clashes with the one used throughout this thesis. Hence, some basic definitions are presented and explained in the following to understand the questions presented in Table 3.3.

To create role-playing objects, first, a role must be instantiated from its role type. This process is called *role creation*. Secondly, with *role playing*, the establishing of the play relationship between a role and a certain player, i.e., an instance of a player type, is denoted. Finally, roles may be targeted for *role movement*. This term denotes the movement of roles within the collection of all roles that are currently played by one object. In contrast to this, a *role transfer* moves roles between different objects.

role creation

role movement

role transfer

Following the life cycle of roles, we analyze the notion of *self*. Typically, self is the initial receiver object of a compound object. Alternatively, with the strategy of *conjunctive attachment*, this compound object is constructed by wrapping roles on their player like layers on an onion [Graversen, 2006]. Here, it does not matter whether the new role is attached directly to the innermost player, or to any of the previously added roles. With this strategy, one of those roles is the initial receiver. In contrast, during role-playing, self may refer to two different objects which is called *dual self*: the initial receiver (the whole compound object) and the currently executing part object (a role instance). Furthermore, the *non-virtual self* addresses the player and their roles as tuples and does not employ compound-object semantics.

self

conjunctive attachment

dual self

non-virtual self

The notion of an *around-method*, being a well-known concept [Moon, 1986], was one out of a large framework of method combinators from Common Lisp Object System (CLOS) [Kleene, 1989]. In the context of aspect-oriented programming, around-methods are used to separate and modularize code crossing inheritance hierarchies. Often crosscutting concerns (e.g., logging, caching, etc.) are implemented with around-methods. They can easily be enabled and disabled during compile- and runtime. This flexibility

around-method

Feature		Description
27.	The amount of roles an instance of a class and a role can play may be constrained.	A.1.1 (p. 155)
28.	Each role type played must be unique.	A.1.1 (p. 155)
29.	Possible supertypes for classes can be class types, role types, or compound object types.	A.1.2 (p. 156)
30.	The amount of simultaneously existing instances of a role type may be constrained.	A.2.2 (p. 157)
31.	The amount of players, a role is played by, may be constrained.	A.2.3 (p. 158)
32.	The visibility of roles during dispatching may be constrained.	A.2.4 (p. 159)
33.	Role types are supertypes, subtypes, or unrelated types of their player.	A.3.1 (p. 160)
34.	Role types may extend role types, class types, or compound objects.	A.3.2 (p. 161)
35.	The player type for a role type may be a role type, a class type, an interface type, a metaclass, a compound object type, a property, or undefined.	A.3.3 (p. 161)
36.	Properties of roles can be fields, methods, class methods, and static methods.	A.4 (p. 162)
37.	Roles can have nested methods, roles and classes.	A.4.4 (p. 162)
38.	Role instances can be referenced directly, or indirectly.	A.5.1 (p. 163)
39.	A reference to a role always points to the compound object.	A.5.1 (p. 165)
40.	Method dispatch on roles happens on the sender or its player, the receiver or its player, the context, or the compound object.	A.5.1 (p. 163)
41.	Self may refer to dual self, or non-virtual self.	A.5.2 (p. 165)
42.	Super refers both to the static inheritance chain, and to the attached roles.	A.5.3 (p. 166)
43.	The player may be referenced directly, or indirectly.	A.5.4 (p. 166)
44.	A role may be called from its player.	A.5.5 (p. 166)
45.	Roles may call among each other.	A.5.5 (p. 166)
46.	Roles may incorporate around-methods.	A.5.6 (p. 166)
47.	Role creation, attachment, and movement may be restricted.	A.7 (p. 168)
48.	Roles may be terminated explicitly, or implicitly.	A.7.3 (p. 170)
49.	Role methods may have various access modifiers.	A.8 (p. 171)
50.	Roles may provide meta-functionality.	A.8 (p. 171)
51.	Roles allow for typed references.	A.8 (p. 171)
52.	Roles may be used as filters.	A.8 (p. 172)
53.	Roles may be used for renaming.	A.8 (p. 172)
54.	Roles may be parameterized.	A.8 (p. 172)

Table 3.3.: Role features solely focusing on runtime aspects.

promotes them for a solution to the composition of roles. Roles may also support instance level composition with around-methods.

role as filter

A *role as filter* is a role providing a unique view on the player [Graversen and Beyer, 2002; Richardson and Schwarz, 1991; Pernici, 1990; Steimann, 2000a; Harrison and Ossher, 1993; Büchi and Weck, 2000; Chernuchin and Dittrich, 2005]. Especially within composition filters [Bergmans and Aksit, 2001], this concept is used. Filters are enabled on incoming, as well as on outgoing messages. Hence, they can be applied on the sender and receiver side. Furthermore, Eiffel contains strong renaming capabilities [Meyer,

1988] to avoid signature clashing. Within role literature [Richardson and Schwarz, 1991; Herrmann, 2005], using a *role for renaming* is mainly motivated to preserve access to information of the player which otherwise would be overridden by the role. Finally, *role parameterization* in terms of Java's generic types [Oracle, 2015] is a fairly new topic in the role literature. It makes roles more type-safe and more reusable.

role for renaming

role parameterization

4

FOUNDATIONS OF DISPATCH

Many developed programming languages offer support for advanced modularization mechanisms, like in the context of this thesis with roles, but are implemented as transformations to the imperative intermediate representation of an already established language (see Sect. 13.2.2). Their core constructs largely overlap in semantics [Kühn and Cazzola, 2016]. Hence, reusing the corresponding transformations requires reusing their syntax as well, which is too limiting.

With *SCROLL*, we identified dispatching as fundamental to role-based programming and propose a declarative and parameterizable approach for four-dimensional, context-aware dispatch at runtime. To increase the modularity of programs, research has introduced different abstraction mechanisms, where one concrete program module does not refer to another concrete module, but only abstractly specifies the functionality or data to be used (*polymorphism*). With roles, this principle is pushed even further. The mechanisms for polymorphism are manifold; they include traditional receiver-type polymorphism and reach out to multiple and predicate dispatching [Ernst et al., 1998], pointcut-advice in aspect-oriented programming [Masuhara and Kiczales, 2003], or layered methods in context-oriented programming [Hirschfeld et al., 2008]. Those languages typically overlap in their semantics but differ syntactically. Compiler frameworks [Ekman and Hedin, 2007; Avgustinov et al., 2006] only support reusing the implementation of a language's execution semantics if that language is extended syntactically.

polymorphism

The process of *dispatching* resolves abstractions and binds concrete functionality to their usage [Bockisch et al., 2012]. This declarative mechanism determines the code to be executed upon a method invocation. It takes place whenever a specific code location is referenced during the program execution. A well-known example of dispatching is *receiver-type polymorphism*. Here, dispatch is choosing the method implementation based on the dynamic type of the receiver object. Languages that go beyond such classic receiver-type polymorphism are called *advanced-dispatching languages*, as they compose functionality in different, more flexible ways and incorporate additional runtime state. Hence, role-based dispatch can be seen as an advanced dispatch suitable to overcome the subtle ambiguities introduced with the concept of role-playing objects. However, role dispatching is a dynamic process. Thus, techniques solely extending the static program structure cannot satisfactorily realize this dynamic process. For example, AspectJ's inter-type declarations can influence the type hierarchy, e.g., by adding interfaces, methods, and fields to a class. While the dynamic semantics of these constructs, namely executing the added methods or accessing the added fields, can be made available for dispatch, their static types inferred by the compiler cannot. Adding an interface changes the type hierarchy. The woven type and its original type differ. The language's type system and the compiler's type checker are not adequately prepared for this weaving process. Whether this impact may be mitigated by making type checkers aware of advanced dispatching with regard to roles, is beyond the scope of this thesis.

dispatching

receiver-type polymorphism

advanced-dispatching languages

As mentioned above, implementations of role-based languages often build on the back-ends of an already established language, thereby reusing the implementation of the constructs in its intermediate language. But not all constructs of role-based languages have a trivial mapping to the established intermediate language (e.g., Java bytecode). The resulting semantic gap between source and intermediate language, i.e., the inability of the intermediate language to directly express the new language's mechanisms, requires compiling the language's high-level concepts down to low-level imperative code. This was considered as inappropriate during the development of *SCROLL*, since building and

maintaining a whole new compiler tool chain is too time-consuming and out of the scope of this thesis.

Advanced compiler frameworks could have been assisting in this task, and even enable to reuse the non-trivial code generation for role-specific language constructs that have no direct counterpart in the target intermediate language. But this reuse requires the new language to be a syntactic extension of an existing one. While code transformations defined on the common intermediate language are shared among all language extensions, they cannot exploit knowledge about new source language constructs. This knowledge is lost during the transformation to the intermediate language. With that, existing tools' usefulness is greatly reduced. The developer has to observe the program execution in terms of the generated, imperative code in the intermediate language rather than in terms of the new language's source-level abstractions. This is considered as a major drawback for existing role-based programming languages. Experimenting with their specific dispatch implementations (e.g., changing and adapting its semantics at runtime) is not possible at all. Thus, we decided to use a library approach with *SCROLL* for role-based, dynamic dispatch, making custom compiler and code generation unnecessary.

Nevertheless, many other programming languages offer such a form of dynamic dispatch. A function consists of a set of implementations, each with a guard specifying the conditions under which that implementation should be executed. When a function is invoked, all the implementations that are applicable, meaning that their guards are satisfied, are considered. For role-based programming, those guards can be derived from the role model itself.

We now focus on dispatch concepts in other mainstream languages. In classical object-oriented languages, such as, Java, a method `m` has an implicit guard specifying that the runtime class of the receiver argument must be a subclass of `m`'s enclosing class. A method `m1` overrides another method `m2` if the enclosing class of `m1` is a subclass of the enclosing class of `m2`. Multimethod dispatch, on the other hand, e.g., found in languages like Cecil [Chambers, 1992] and MultiJava [Clifton et al., 2000] abstracts from those implicit guards to support runtime class tests on basically any subset of the method's arguments. The overriding relation is generalized to all of its arguments as well. Pattern matching in functional programming languages (e.g., in ML [Milner, 1997]) falls into that category of dispatch as well, allowing guards to test the datatype constructor of arguments and to recursively test the substructure of arguments. In this approach, the textual ordering of function implementations determines the overriding relation. Overall, dynamic dispatch offers many advantages over manual dispatch using if statements:

Declarative specification The guards on each method implementation can be specified declaratively. Then, the "best" implementation is selected for a given invocation.

Inheritance support Dynamic dispatch makes functions extendable via inheritance. Functions may be extended by writing additional implementations handling new scenarios. No existing code has to be modified.

Type checking Static type checking is enhanced in comparison of using if statements and doing runtime type-casting manually. With that improvement, method lookup cannot fail as there are no dynamic *message not understood* or *message ambiguous* errors anymore.

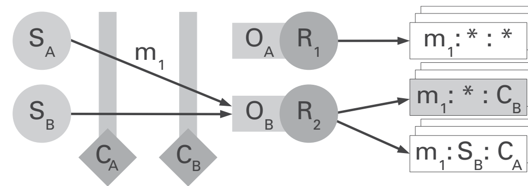


Figure 4.1.: The concept of four-dimensional dispatch (with methods m_n, sender S_n, contexts C_n, and objects O_n with roles R_n) [Hirschfeld et al., 2008].

4.1. PREDICATE DISPATCH

As an advanced form of dynamic dispatch, the concept of predicate dispatch was introduced [Chambers, 1992; Ernst et al., 1998]. With this, a method implementation may specify an arbitrary predicate as a guard. Consequently, a method `m1` overrides another method `m2` if `m1`'s predicate logically implies `m2`'s predicate. This concept unifies and generalizes several existing language concepts, including object-oriented dynamic dispatch, multimethod dispatch, and functional-style pattern matching. Nevertheless, predicate dispatch (and its ongoing work on top, e.g., with Chambers and Chen [1999]; Ucko [2001]; Orleans [2002]) has several issues with regard to its utility in practice:

Actual implementations Implementations of predicate dispatch are only surveyed in the context of non-mainstream programming languages or toy examples, not including static type checking.

Type checking The static type checking for predicate dispatch as presented in Ernst et al. [1998] requires a type system with access to the whole program. This clashes with the modular type checking style of mainstream object-oriented and thus with role-based programming languages. Hence, specifying basic well-formedness properties for individual classes or roles, remains difficult.

Conservativeness Other more static approaches are overly conservative, ruling out many desirable uses of predicate dispatch. For instance, the resulting type system cannot infer that the predicates $x > 0$ and $x \leq 0$, where `x` is an integer argument to a function, are exhaustive and mutually exclusive. Functions consisting of two implementations with these guards will be rejected throwing both exhaustiveness and ambiguity errors.

Pratical use Finally, little to no evidence has been presented to show the utility of predicate dispatch in real-world applications.

With those limitations for predicate dispatch in mind, we decided to use a basic, more grounded approach to tackle the ambiguities introduced with roles. With function composition and pattern matching style dispatch guards, we end up with a declarative and parameterizable approach for four-dimensional, context-aware dispatch at runtime in *SCROLL*. This is presented in more detail in Sect. 9.3.

4.2. MULTI-DIMENSIONAL DISPATCH

In the following, the concept of multi-dimensional message dispatch [Hirschfeld et al., 2008] in the context of role-playing objects is presented. It is intended to help the reader to understand its basics for the remainder of this thesis. This can be derived from:

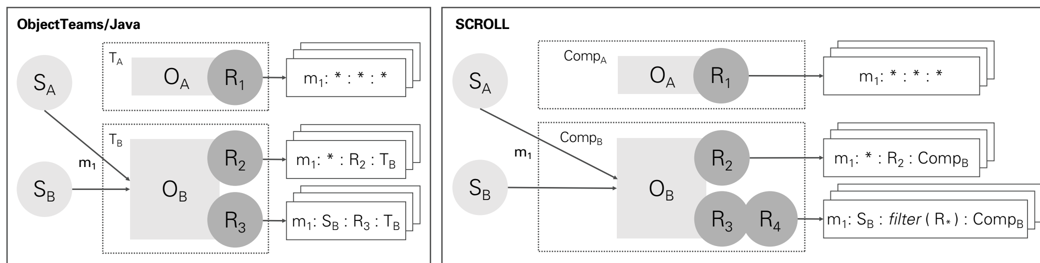


Figure 4.2.: Multi-dimensional Dispatch with OT/J and *SCROLL* in comparison (with methods m_n, sender S_n, teams T_n, compartments $Comp_n$, and objects O_n with roles R_n).

One-dimensional dispatch Classical C-style procedural programming only offers the static binding of a function with a name. Calls are directly mapped to the corresponding implementations. Calling a method `m` leaves no choice but the invocation of its only implementation of method `m`.

Two-dimensional dispatch With object-oriented programming, the second dimension was added. In addition to resolving the method name, the receiver of the call is taken into consideration when looking up that method.

Three-dimensional dispatch For instance, subjective programming [Smith and Ungar, 1996] extends the object-oriented method dispatch by yet another dimension. With that, the targeted method is not only selected in dependence of its name and receiver, but also in dependence of the sender.

Four-dimensional dispatch As part of this thesis, the fourth layer is put on top of the dispatching concepts, inspired by subjective and context-oriented programming [Hirschfeld et al., 2008]. Now also the context of the actual message send, hence the overall system's context, is taken into account (see Fig. 4.1). Based on that information, methods or their partial definitions are selected or excluded from the message dispatch. This finally enables the context-dependent behavioral adaptation and variation needed for role-based programming.

To provide a more fine-grained dispatch, role-oriented programming languages, such as, OT/J, encapsulate the fourth dimension within contexts implemented as first-class citizens (e.g., in OT/J these contexts are called Teams). With that, the actual method dispatch is configurable at compile-time for a predefined set of role types (see Fig. 4.2 left side). *SCROLL* builds conceptually on-top of that. It allows for the dynamic attachment of arbitrary many roles at runtime, still encapsulated in first-class contexts, called compartments, as already introduced (see Sect. 3). Now, the method dispatch can be re-configured at runtime via filtering the set of all attached roles steered by user-defined functions (see Fig. 4.1 right side). This is explained in more detail in Sect. 9.3.

In sum, the dispatch mechanism provided by *SCROLL* is declarative (and structurally attached to context specifications incorporated as first-class citizens), and can be parameterized via user-defined filter and sorting functions. This allows for four-dimensional dispatch within structured contexts. Instead of only associating the behavior called with a name (first dimension), the receiver context (second dimension), the sender context (third dimension), and the overall system context (fourth dimension, but now structured) are taken into account, while being re-configurable at runtime.

5

FOUNDATIONS OF GRAPHS AND GRAPH FILTERING

Graphs are one of the most important data structures in programming and computer science in general. They appear in almost all applications. Structures linked with software models, pointers, object nets, databases, and with various schemes, are in essence graphs. A labeled graph is a collection of objects (nodes or vertices) which are connected via linking objects (edges). Nodes and edges may be associated with names, additionally (called labels). Graphs are applied as background data structure in *SCROLL*. The real power of graphs makes itself apparent when traversing multiple steps in order to unite disparate not directly connected vertices by a path. The type of path taken, defines the higher order, inferred relationship that exists between two vertices. Paths form the core of the presented graph traversals. A traversal refers to visiting elements (i.e., vertices and edges) in a graph in some algorithmic fashion. A *graph* is defined as:

graph

Definition 6 ((Labeled) Graph)
In general, a (labeled) graph H is a 4-tuple (V, E, Lab, L_Σ) with:

- V is a finite set of vertices (nodes) with $|V| \geq 0$,
- E is the set of edges, where E is a relation $E \subseteq V \times V$,
- L_Σ is the set of labels and
- $Lab : V \cup E \rightarrow L_\Sigma$ is the labeling function, which assigns a label to each node in V and edge in E.

Such definitions are usually sufficient for deriving theorems. However, in the scope of this thesis, where the graph is required to be embedded handling the associations of role-playing objects, this definition says nothing about a graph's realization. Nevertheless, graphs for storing data and their relationships offer a significant advantage. There is a constant time cost for retrieving an adjacent vertex or edge. Regardless of the size of the graph, the cost of a local read operation at a vertex or edge remains constant [Rodriguez and Neubauer, 2010]. This benefit is so important that it creates the primary means by which users interact with graph traversals. Graphs offer a unique vantage point on data, where the solution to a problem is seen as abstractly defined traversals through its vertices and edges. The implementation of a graph determines the efficiency of the operations that are applied to it. Exactly those efficient graph operations yield an unconventional problem-solving style. This style of interaction is dubbed the graph traversals in the following and forms the primary point of discussion for this section.

The functional, flow-based approach to traversing graphs and different types of traversals over different types of graph data sets, supports different types of problem solving processes [Rodriguez and Neubauer, 2010]. The most primitive, read-based operation on a graph is a single step traversal from element i to element j, where $i, j \in (V \cup E)$. For example, a single step operation can answer questions such as "which edges are outgoing from this vertex?" or "which vertex is at the source of this edge?". Single step operations expose explicit adjacencies in the graph. Various types of those single step traversals can be found in Table 5.1. These operations are defined over power multiset domains and ranges. This naturally allows for function composition. When edges are labeled and elements have properties, it is desirable to constrain the traversal to edges of a particular label or elements with particular properties. These operations are known as

Notation	**Meaning**
Basics	
$f : D \to R$	Function signature for function f with domain D and range R
$f(a_1, a_2, ...)$	Function application with arguments a_1, a_2 etc.
$\circ$	Path composition
$\mathcal{P}(A)$	Power set of set A, set of all subsets of A (i.e., 2^A)
$\hat{\mathcal{P}}(A)$	Power multiset of A, infinite set of all subsets of multisets of A
Traversals	
$\mathscr{E}_{out} : \hat{\mathcal{P}}(V) \to \hat{\mathcal{P}}(E)$	Yield all outgoing edges of a multiset of vertices
$\mathscr{E}_{in} : \hat{\mathcal{P}}(V) \to \hat{\mathcal{P}}(E)$	Yield all incoming edges of a multiset of vertices
$\mathscr{V}_{out} : \hat{\mathcal{P}}(E) \to \hat{\mathcal{P}}(V)$	Traverse the outgoing (i.e., sink) vertices of the edges
$\mathscr{V}_{in} : \hat{\mathcal{P}}(E) \to \hat{\mathcal{P}}(V)$	Traverse the incoming (i.e., source) vertices of the edges
$\epsilon : \hat{\mathcal{P}}(V \cup E) \times R \to \hat{\mathcal{P}}(S)$	Get the element property values for key $r \in R$
Filters	
$\mathscr{E}^{\sigma}_{lab\pm} : \hat{\mathcal{P}}(E) \times \Sigma \to \hat{\mathcal{P}}(E)$	Allow (+) or filter (-) all edges with the label $\sigma \in \Sigma$
$\epsilon_{p\pm} : \hat{\mathcal{P}}(V \cup E) \times R \times S \to \hat{\mathcal{P}}(V \cup E)$	Allow (+) or filter (-) all elements with the property $s \in S$ for key $r \in R$
$\epsilon_{\epsilon\pm} : \hat{\mathcal{P}}(V \cup E) \times (V \times E) \to \hat{\mathcal{P}}(V \cup E)$	Allow (or filter) all elements that are provided elements

Table 5.1.: Notation overview for graph traversals and filters.

filters

filters and are abstractly defined in Table 5.1 as well. Through function composition of single step traversals, we can define graph traversals parameterized by filter functions which is suitable to answer questions, such as, "which roles is the current object playing?". This is demonstrated with the following example traversal and filtering. If i is the vertex representing the role-playing object and

$$f : \hat{\mathcal{P}}(V) \to \hat{\mathcal{P}}(S),$$

where

$$f(i, \text{ name}) = \epsilon(\mathscr{V}_{in}(\mathscr{E}_{lab+}(\mathscr{E}_{out}(i), \text{ plays})), \text{ name}),$$

then $f(i, \text{ name})$ will return the property *name* of the roles that the object in question is playing. Through function composition, the previous definition can be represented more clearly with the following function rule:

$$f(i, \text{ name}) = (\epsilon^{\text{name}} \circ \mathscr{V}_{in} \circ \mathscr{E}^{\text{plays}}_{lab+} \circ \mathscr{E}_{out})(i, \text{ name}).$$

This function f says, traverse to the outgoing edges of vertex i representing the role-playing object, then only allow those edges with the label *plays*, then traverse to the incoming (i.e., source) vertices on those *plays*-labeled edges. Finally, return the property name of those vertices. The function f is a *higher-order adjacency* defined as the composition of explicit adjacencies and serves as a join of the role-playing object and its roles name properties. Those traversals, parameterized by filter functions, allow for the

higher-order adjacency

filtering of a single graph data structure. The parameterization alters the semantics of the final query. In consequence, this enables the altering of the dispatching semantics in role-based applications.

Graphs are a flexible modeling construct that can be used to model a domain and partition that domain into an efficient, searchable space. When the relations between the objects of the domain are seen as vertices, then a graph is simply an index that relates vertices to vertices by edges. The way in which these vertices relate to each other determines which graph traversals are most efficient to execute and which problems can be solved by the graph data structure. For many problems, only local subsets of the graph need to be traversed to yield a solution which is handy in the context of roles. By structuring the graph in such a way as to minimize traversal steps, limit the use of external indices, and reduce the number of set-based operations, graphs provide a flexibility that is difficult to accomplish with other data structures.

6

A MOTIVATING EXAMPLE FOR MULTI-DIMENSIONAL DISPATCH

Haddadin et al. [2011] focus on the idea of a human and a robot working together in an intimate collaboration. They propose an approach to combine human and robot capabilities for executing certain tasks in a partially unknown environment. Using sensor networks a safe integration of humans and robots for a specific task execution can be achieved. This integration of humans and robots can be described by the states of a hybrid automaton. Transitions are based on sensor inputs enabling the robot to react to unexpected asynchronous events. Additionally, safety is of fundamental concern if human-robot cooperation shall ever be realized in industrial practice. This section is not concerned with the mechanical design of robots or to make them sufficiently safe at all. Instead, this example is selected to demonstrate the use of role-based design and implementation because it inherently exposes role orientation.

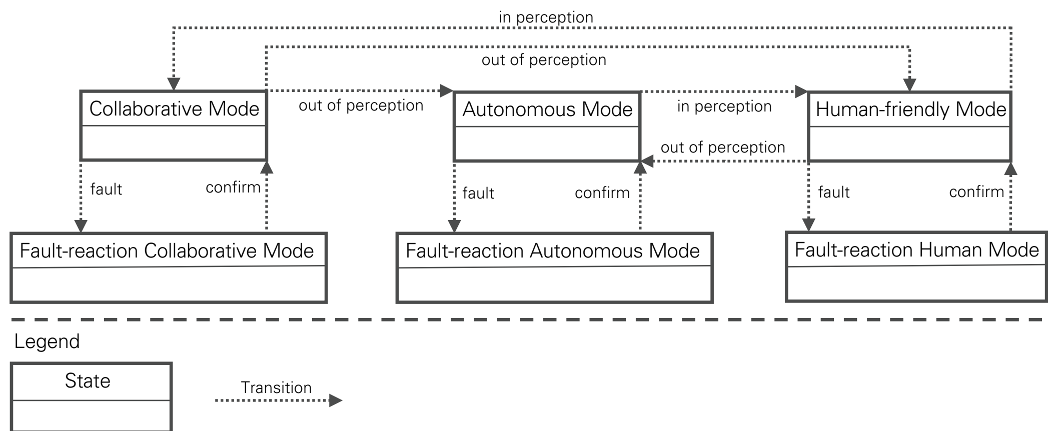


Figure 6.1.: The Haddadin automaton adapted from Haddadin et al. [2011].

In current state of the art for industrial workflows in manufacturing execution systems, simple sequences of tasks with static execution orders and binary branching are used. Fault tolerance is not a problem due to specially crafted working environments. Especially human-robot interaction is not yet safe and legal foundations are missing [Haddadin et al., 2011]. With the aforementioned automaton, an integrative and flexible approach to carry out the desired task in a very robust yet efficient way was presented. It enables to distinguish between different collaboration and fault stages, which slow down or stop the entire process. Interruptions within execution steps are part of the concept and do not require some special treatment. In order to optimally combine human and robot capabilities, the robot must be able to adapt to the human intention during task execution to achieving safe interaction. Thus, the measurement of human state is the dominant factor influencing the transition between the proposed functional modes. As modeled in Fig. 6.1, the human state is primarily used to switch between different functional modes of the robot. We distinguish between four major functional modes of the robot in a co-worker scenario [Haddadin et al., 2011]:

Autonomous task execution The autonomous mode in human absence.

Human-friendly behavior The autonomous mode in human presence.

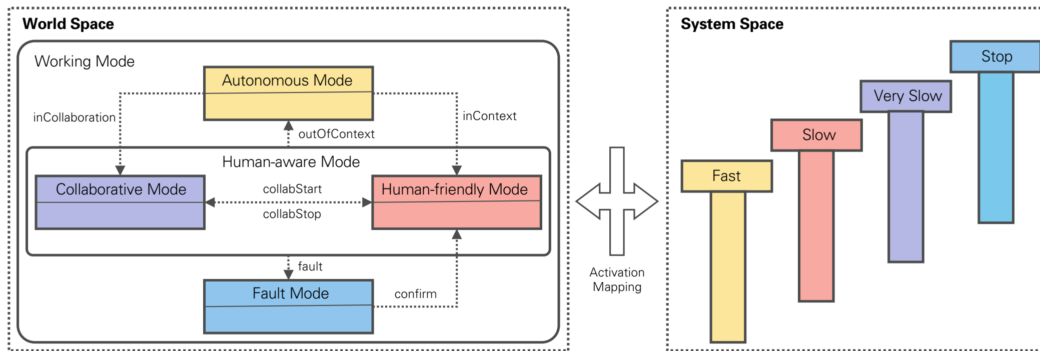


Figure 6.2.: The mapping from the Haddadin world space to the system space.

Co-Worker behavior The cooperation with human in the loop.

Fault reaction behavior The safe fault behavior with and without human in the loop.

The conditions for transitions between these states provide high flexibility in the application design. In the *first* functional mode, the robot is autonomously fulfilling its given task without considering the human presence. In the *second* and *third* modes, we need a meaningful partition of the task space which subdivides the given workspace of the robot into regions of interaction. For the sake of simplicity, we decided to solely depend the robots speed, i.e., a single variable, for the different functional modes. Thus, the activation mapping from states to roles as shown in Fig. 6.2 are used (indicated by colors).

In the *third* mode, interaction tasks are carried out for fulfilling a common desired goal, involving a collaboration of human and a robot. These two modes allow for the seamless switching between each other. A fault reaction behavior is represented with the *fourth* mode, addressing the reaction to stop it self if a fault occurs.

We implement a simple robotic co-working system as proof of concept in which role-based programming is applied to dynamically adapt the system behavior at runtime. How could a practical role-based implementation of the aforementioned functional modes for a maximum of flexibility and adaptivity look like? We derive important requirements for the design of *SCROLL* from the identified problems. For the implementation of the robotic co-working example, the software design needs to contain definitions and descriptions to handle the static and dynamic aspects of problems in this domain, namely, for business logic as well as adaptation logic. The interactions of the participating objects with the client are non-trivial. Each of those clients will handle the instances differently and with different use-cases in mind, e.g., handling different kinds of robots and their individual work plans. This interaction needs to be described and constrained from the viewpoint of all relevant contexts. Furthermore, a sound understanding of object collaborations and relationships at runtime is crucial.

When developing an adaptive software system, the programmer must be able to define those collaborations. Once they are understood, it is important to group and separate them for the sake of comprehensibility and reuse. Relationships need to be assigned to relevant contexts. Derived from the two aforementioned tasks, this reuse is one of the most important attributes adaptivity on the resulting system benefits from. With invariants and constraints, the behavior of the objects working together in that systems can be restricted and checked during runtime.

Finally, it is impossible to foresee each and every possible future use-case and context. Listing 6.1 provides an excerpt for a plain, simple object-oriented implementation of robotic co-working example in Scala. Handling the adaptation based on various events

with regard to the current system context and state leads to *if-bloating* with nesting and inter-tangling of business and adaptation logic. The problems mentioned above and introduced in Chapter 3 manifest themselves as:

if-bloating

Increased class complexity (/P.1/) Using a class with inter-tangled implementations of business and adaptation logic depending on various use cases and contexts will require a lot of additional management code and the application of glue patterns. With that, maintainability, extendability, and testability will suffer.

No first-class object collaborations (/P.2/), low separation of concerns (/P.3/) The collaborations between participating objects is not described as first-class citizens, but interleaved and tangled across the resulting, overall implementation.

Lack of reuse (/P.4/) Thus, reuse, maintainability, and extendability is greatly reduced.

No explicit invariants and constraints (/P.5/) Additionally, context checks between different parts of the adaptation logic cannot be specified explicitly but will be spread over the implementation as hard-coded, additional and potentially deeply nested if-blocks.

Lack of adaptivity (/P.6/) In consequence, adaptivity of the resulting software system suffers and is harder to maintain and extend for future unforeseen use-cases and application contexts.

```
1  def handleCoWorking(): Unit = {
2    while(true) {
3      if(currentState == AUTONOMOUS) {
4        if(event.isInstanceOf[InPerception]) {
5          speed = 30; currentState = HUMANFRIENDLY
6        }
7        if(event.isInstanceOf[Fault]) {
8          speed = 0; currentState = FAULTREACTIONAUTONOMOUS
9        }
10     }
11     if(currentState == HUMANFRIENDLY) {
12       if(event.isInstanceOf[InPerception]) {
13         speed = 20; currentState = COLLABORATIVE
14       }
15       if(event.isInstanceOf[OutOfPerception]) {
16         speed = 50; currentState = AUTONOMOUS
17       }
18       if(event.isInstanceOf[Fault]) {
19         speed = 0; currentState = FAULTREACTIONHUMAN
20       }
21     }
22     /* ... */
23   }
24 }
```

Listing 6.1.: Excerpt for a plain, simple implementation of robotic co-working example. Handling the adaptation based on various events with regard to the current system context and state leads to if-bloating with nesting and inter-tangling of business and adaptation logic.

Applying the concept of roles now enables explicit separation of concerns of the relationships at instance level. This cannot be achieved with normal class interfaces. A role

type exactly specifies how the instance of a class interacts in a certain context. The major part of the complexity of those adaptive software infrastructures stems from complex object collaborations. These collaborations become manageable with the break-down into individual, smaller role models. Each role model describes an individual aspect of the object collaboration and adaptation. With the resulting clear separation of object collaboration into smaller pieces with regard to the concerns of the problem space and their composibility, a high amount of reuse is enabled. Furthermore, a role type is a good target for invariants and constraints. And finally, adding role types dynamically allows for the adaptation of unforeseen contexts. The expressiveness of the concept of roles require a whole new level of dispatching semantics, i.e., a dispatch that needs to support context-awareness and dynamically evolving objects on the instance level.

7

RESEARCH CHALLENGES

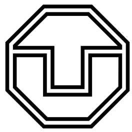
Problem	Requirement	Description
	Functional	
/P.4/	/F.1/	No additional tooling
/P.6/	/F.2/	Dispatch configurable at runtime
/P.2/, /P.3/, /P.5/, and /P.6/	/F.3/	Handle multi-dimensional dispatch
	/F.3.1/	Associate the computational unit with a name
	/F.3.2/	Take the receiver context into account
	/F.3.3/	Take the sender context into account
	/F.3.4/	Take the system context into account
/P.1/, /P.2/, and /P.3/	/F.4/	Increase modularity through role-based programming
	(Semi-) functional or non-functional	
/P.2/, /P.3/, /P.4/	/S.1/	Declarative and parameterizable dispatch description
/P.4/	/S.2/	Easy to use programming model and API
	/S.3/	Reasonable performance / scalability
/P.4/	/S.4/	Integration in existing tool-chains
/P.4/	/S.5/	Integration / compatibility with existing legacy code
	/S.6/	High maintainability
	/S.7/	High extensibility

Table 7.1.: Functional and non-functional requirements for *SCROLL*.

The following research challenges and requirements can be derived from the aforementioned problems (Sect. 6). We aim for a solution that requires no additional tooling (/F.1/). As almost all the contemporary approaches for role-based programming use custom compilers or code generators, they break with existing tool-chains (e.g., debuggers) and cannot be used with ease in widely established integrated development environments (e.g., Eclipse or IntelliJ). This hinders maintainability and extendability and often results in abandoned projects. With the notion of roles, their expressiveness and their subtle ambiguities, the resulting dispatch semantics needs to be handled explicitly and at runtime (/F.2/). Hence, the solution developed during this thesis should support this new kind of context-aware dispatch on the instance level for dynamically evolving role-playing objects. As this context-aware dispatch introduces additional dimensions, those must be addressed as well (/F.3/). Thus, on-top of associating the method with a name and tailoring the dispatch to the receiver or sender context, the overall system context needs to be addressed additionally. And finally as a fourth requirement, a maximum of the features of roles in role-based programming has to be supported by the developed implementation to increase modularity (/F.4/).

Furthermore, some (semi-) functional or non-functional requirements are important as well. To make the aforementioned dispatch implementation as usable and attractive to the developer it should be declarative and parameterizable at runtime (/S.1/). This

offers high flexibility for context-aware adaptation and additional separation of concerns. To support the application programmer even better, the programming model and API should be easy to use and readable even for inexperienced developers (/S.2/). A reasonable performance and scalability of the implementation is important as well for real-world scenarios and show its practical applicability (/S.3/). As a follow-up to /F.1/, code should be easily manageable by existing tool-chains so that future role researchers, i.e., researchers interested in role-based programming, and application developers can continue the development and provide extensions and adaptation for future use-cases and scenarios (/S.4/). For easier integration of existing software systems, we do not want to impose additional burden to the developer writing adapters, proxies, or management code to integrate existing legacy code (/S.5/). Finally, and as a result of /F.1/, /S.2/, /S.4/, and /S.5/, the reference implementation developed during this thesis for role-based programming and dispatch should be highly maintainable and extendable so that future researchers have an easy time providing modifications for new use-cases and scenarios (/S.6/, and /S.7/). A summary can be found in Table 7.1 and will be used as success criteria throughout this thesis (see Sect. 11.1.1).



TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software- and Multimedia-Technology

Software Technology Group

PART III.

ROLES IN STRUCTURED CONTEXTS AT RUNTIME

8

THE EMBEDDED DSL *SCROLL*

"Static typing where possible, dynamic typing when needed!"

Meijer and Drayton [2004]

Scripting languages like Python, JavaScript, Ruby, Perl or Lua offer a flexible object semantic to the developer. On the one hand, programmers can rely on classical object-oriented features, such as inheritance, encapsulation and polymorphism, and on the other, they are able to add and remove members from existing objects or merge them at any given point in their life-cycle [Menon et al., 2013] which is usually not available in statically typed object-oriented languages. Imagine you have a client object that wants to execute some behavior at an object of interest but that desired behavior is not available (Fig. 8.1). Unfortunately, using inheritance, mixins and traits or adapting design-patterns has many disadvantages. The first three techniques will result in a very static system design and exponentially many classes, while the use of patterns often leads to split-objects and the need of additional management code. Adding and removing members from existing objects at runtime are indeed very useful operations for modern software-systems that have a very high demand for adaptivity and need to cope with complexity and change [Furrer, 2015]. Is bridging the gap between statically-typed, object-oriented languages and roles as evolving objects at runtime possible without too much effort? The main contributions of this chapter permit us to answer this questions positively:

SCROLL and the _SCROLL_ MOP An overview on *SCROLL*, an embedded DSL and its underlying MOP [Kiczales et al., 1991; Mernik et al., 2005] that allows for the pure embedding [Hudak, 1998] of roles in the modern, statically typed object-oriented language Scala. It solely utilizes features that are available through the standard compiler. The library allows for easy integration of legacy code and a high separation of concerns. It is limited with regard to type-safety as one might expect. Nevertheless, having a statically-typed host language for roles supports the developer with the best of both worlds: static typing leads to an earlier detection of programming mistakes through static code analysis, better documentation in form of type-signatures, compiler-optimization, runtime-efficiency and an improved design-time development experience, while the latter supports easy prototyping, change to unknown requirements or unpredictable data and application integration. Essentially, two user groups for *SCROLL* can be identified: the *end-user* (the programmer writing domain-specific, role-based applications), and the *library developer* (adapting or transferring the *SCROLL* MOP and its semantics to his research area). For those library developers, the *SCROLL* MOP is discussed in more depth in Chapter 9.

end-user

library developer

Simplicity Based on three concepts (compiler rewrites, implicit conversions, and a definition table), an implementation pattern is presented.

Examples Finally, example applications show how roles are realized with *SCROLL*.

Scala was chosen as host language for *SCROLL*, not only because of its combination of object-oriented and functional programming features, but as well due to its scalability and interoperability with the Java virtual machine providing easy integration of legacy code and availability of already established tools. *SCROLL*, in particular, takes advantage of Scala's features such as higher order-functions, general operator notations, flexible syntax, implicits, compiler rewrites and implicit definitions of parameters.

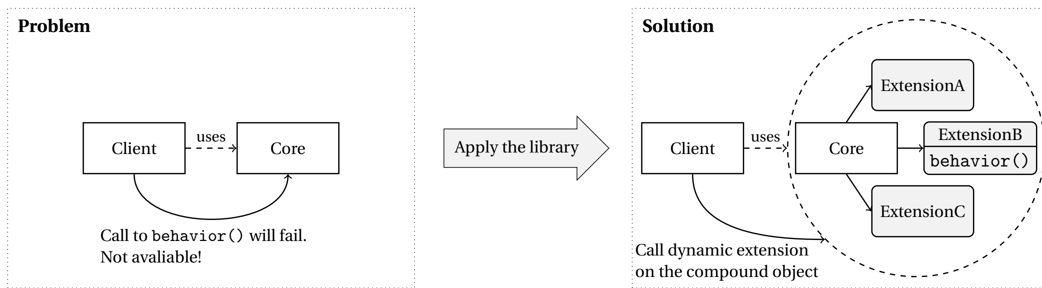


Figure 8.1.: A client wants to execute behavior at a (core-) object but the requested behavior is not available (*left box*). Applying *SCROLL* allows for dynamically adding new behavior at runtime while wrapping all the extension parts (`ExtensionA`, `ExtensionB` and `ExtensionC`) of the augmented object (`Core`) into one *compound object* (*right box*).

8.1. THE BASIC INGREDIENTS OF THE EMBEDDED DSL *SCROLL*

SCROLL is an embedded method-call interception DSL [Lämmel, 2002; Mernik et al., 2005] tailored to the features needed to implement roles and resolve the ambiguities arising with regard to dynamic dispatch. The library approach together with an implementation with Scala was chosen for mainly the following reasons: it allows focusing on role semantics, supports a customizable, dynamic dispatch at runtime, and allows for a terse, flexible representation. No additional tooling (like a custom lexer, parser or compiler) is needed to execute the *SCROLL* MOP. It is purely embedded in the host language, thus uses the standard Scala compiler to generate Java Virtual Machine (JVM) bytecode. With that, the implementation is reasonable small (~1400 lines of code) and maintainable. The programming interface with Scala's flexible syntax holds the property of being easily readable, even to inexperienced users. We have taken a layered approach (see Fig. 8.2) for designing and implementing *SCROLL*:

Usage Layer This is the end-user layer, tailored for the instantiation and use of objects with their roles as dynamic extensions forming evolving objects. Role objects, as well as their enclosing compartments, may be instantiated from standard Scala classes, case classes, or traits.

Configuration Layer All role-specific features are aggregated into the `Compartment` trait and its utility traits (e.g., `DispatchQuery`, `RoleConstraints`, or `RoleGroups`). They implement the full interface of *SCROLL* and are configurable at runtime through concrete instances. Altering their default behavior is viable via subclassing. This layer is targeted to both end-users and library developers.

MOP Layer This layer contains the implementation of the metaclasses `Compartment` and its helper traits (i.e., the MOP). Especially the dynamic dispatch semantics within the `DispatchQuery` trait are targeted to be investigated and adapted by library developers.

Specification Layer To handle the actual dispatching on the compound object, this layer contains specifications for the dispatch (`SCROLLDispatch`, `SCROLLDynamic`). This unifies their complex semantics into only two interfaces rather than scattering them across many interfaces. This layer should be changed if a library developer wants to change the semantics of the dynamic dispatch within *SCROLL*.

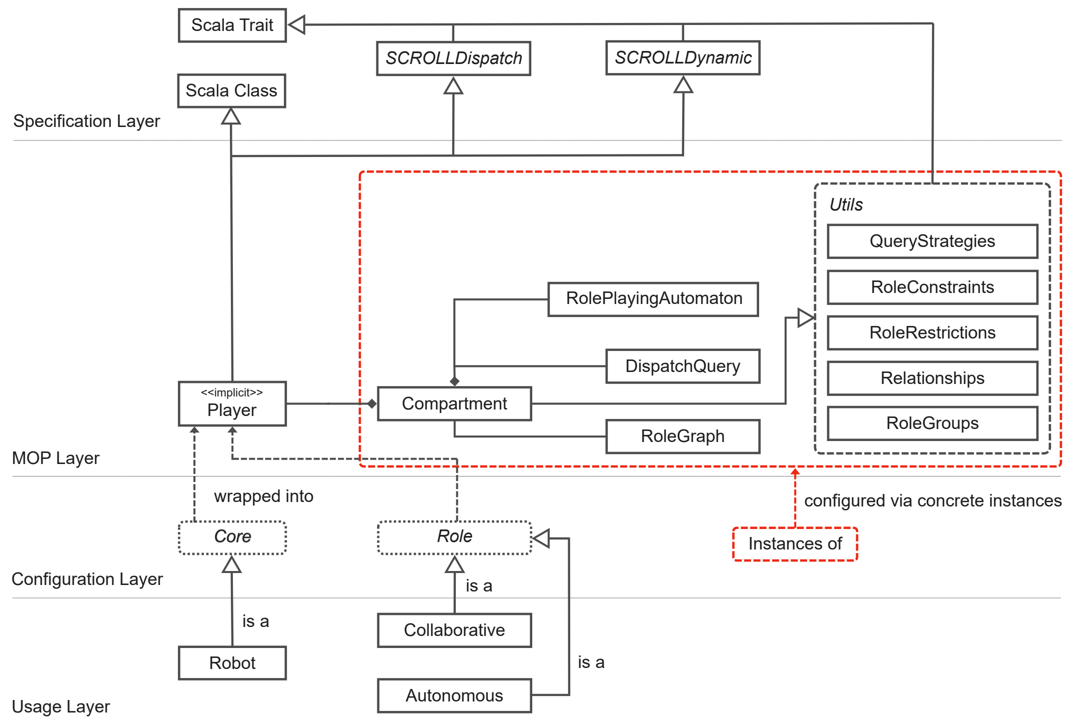


Figure 8.2.: An overview of the *SCROLL* metamodel and MOP layers.

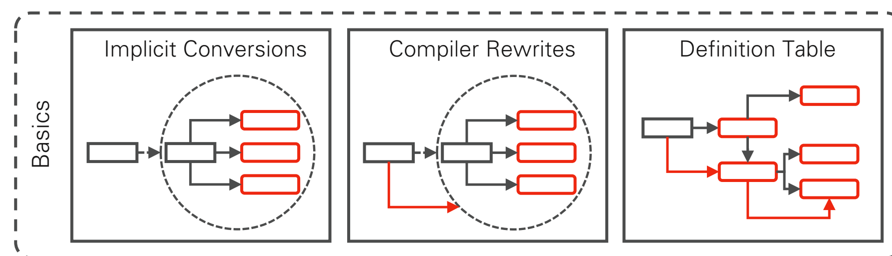


Figure 8.3.: Required basics for the implementation of a DSL for roles in structured contexts at runtime.

We start with describing the three basic implementation concepts required from the host language to provide a pure embedding of roles in structured contexts (Sect. 8.2). Afterwards, the four layers themselves are explained in more detail. First, by presenting the usage layer (Sect. 8.3), followed by the remaining ones (configuration in Sect. 9.1, MOP in Sect. 9.2, and specification in Sect. 9.3).

8.2. BASIC IMPLEMENTATION CONCEPTS

To provide a DSL for the pure embedding of roles in structured contexts, *SCROLL* requires the following basic implementation concepts from the host language (see Fig. 8.2):

Compiler rewrites A concept for compiler rewrites for method calls, functions calls, and attribute access is required. It hands over calls to the library for finding behavior and structure that is not natively available at the player. This can be seen as a compiler-supported variant of method-call interception [Lämmel, 2002].

Implicit conversions For aggregating the compound object from the core and its roles,

implicit conversion

and for exposing the *SCROLL* MOP API, implicit conversions are needed. An *implicit conversion* from type `S` to type `T` is defined by an implicit value which has the function type `S => T`, or by an implicit method convertible to a value of that type (see Appendix B). Implicit conversions are applied in two situations: i) If an expression `e` is of type `S`, and `S` does not conform to the expression's expected type `T`, and ii) in a selection `e.m` with `e` of type `S`, if the selector `m` does not denote a member of `S`. In the first case, a conversion `c` is searched for which is applicable to `e` and whose result type conforms to `T`. In the second case, a conversion `c` is searched for which is applicable to `e` and whose result contains a member named `m`.

definition table

Definition table for the plays relationship The relationships between each individual player and its roles need to be stored. A *definition table* holds all kinds of program components, whose attributes are created by declaration: types, variables, methods, functions, and parameters [Waite and Goos, 2012]. In *SCROLL*, a definition table for roles is implemented with a graph-based data structure, but it may be implemented with tables, maps, or lists as well.

If one is able to find or emulate these three techniques in the desired host language, it is easy to provide an alternative implementation of *SCROLL*. In the following, these basic concepts are explained in more detail.

8.2.1. THE DYNAMIC TRAIT WITH COMPILER REWRITE RULES

Behavior and state of roles that is not natively available at the player needs to be addressed somehow. Scala's `Dynamic` trait can be used to implement that behavior [EPFL, 2016b]. To get invoked, the proper role has to be identified and selected. To do so, calls to role-specific functionality that would normally fail during type checking phase, are rewritten by the compiler according to the rules shown in Listing 8.1. This transformation is type-unsafe, because the actual set of roles as dynamic extensions that are bound to the player, is not statically known. Hence, static type-safety is not available. As an example, the method call to the robots `name` attribute from Listing 8.7 (Line 7) is translated as presented in Listing 8.2.

```
foo.method("param")        ⇝ foo.applyDynamic("method")("param")
foo.method(x = "param")    ⇝ foo.applyDynamicNamed("method")(("x", "param"))
foo.method(x = 1, 2)       ⇝ foo.applyDynamicNamed("method")(("x", 1), ("", 2))
foo.field                  ⇝ foo.selectDynamic("field")
foo.varia = 10             ⇝ foo.updateDynamic("varia")(10)
foo.arr(10) = 13           ⇝ foo.selectDynamic("arr").update(10, 13)
foo.arr(10)                ⇝ foo.applyDynamic("arr")(10)
```

Listing 8.1.: Compiler rewrite rules from the `Dynamic` trait [EPFL, 2016a].

```
+this.name                 ⇝ this.unary_+().name
                           ⇝ new Player[Robot](this).name
                           ⇝ new Player[Robot](this).selectDynamic("name")
```

Listing 8.2.: Rewriting for dynamically rewritten access to the `Robot` attribute `name`.

SCROLL hooks into those rewritten calls and triggers the actual invocation of the appropriate roles, as well as the error handling. It refrains from using runtime exceptions or similar exception-based error handling in case of not being able to find the functionality the developer is querying for. Instead, Scala's `Either` container type is applied. It has two sub-types, `Left` and `Right`. If an `Either[A,B]` object contains an instance of `A`,

then the `Either` is a `Left`. Otherwise, it contains an instance of `B` and it is a `Right`. By convention, it is used to carry the error case as `Left` (e.g., `DynamicBehaviorNotFound`), whereas the `Right` contains the success value (e.g., the result of executing the dynamic behavior). Together with a sealed type hierarchy with data types using case classes that represent errors, very readable messages compared to actual stack-traces from standard Java exceptions are generated.

8.2.2. BOXING WITH IMPLICITS

We want to be able to add roles to any given object of any type in Scala. Implicit conversions [Odersky et al., 2008] provide a lightweight way to expose *SCROLL*'s API for adding, removing and transferring behavior or state to any object and is implemented via the class `Player` from the *SCROLL* MOP layer. The following code listing gives an excerpt:

```
1  implicit class Player[T](val wrapped: T) {
2    /* Applies lifting to Player */
3    def unary_+ : Player[T] = this
4
5    def play(role: Any): Player[T] = /* ... */
6
7    def drop(role: Any): Player[T] = /* ... */
8
9    def transfer(role: Any) = new {
10     def to(player: Any) { /* ... */ }
11   }
12
13   /* ... */
14
15   override def equals(o: Any) = /* ... */
16 }
```

Listing 8.3.: The generic `implicit` class `Player` from *SCROLL*'s MOP layer.

Scala's implicit conversion is used to wrap the player into an equivalent compound object exposing the required API in a type-safe manner. Furthermore, the issue of object schizophrenia needs to be addressed with a clear notion of object identity. The identity of an object should be the same independent of which role is attached. In summary, four kinds of equality tests between pairs of objects (i.e., the player C and its roles R_n) are possible:

1. $C + R_1 == C$
2. $C + R_1 == C + R_1$
3. $C + R_1 == C + R_2$
4. $C == C + R_1$

To overcome object schizophrenia for equality tests in *SCROLL*, the library modifies the identity-related method of the compound object represented by `Player` as shown in the above code-listing (Listing 8.3). In fact, == and the `equals`-method are equivalent in Scala that is, the expressions `x == y` and `x.equals(y)` give the same result. We define the `equals`-method in the following ways:

1. $C + R_1 == C$: When the equality for a player playing a role compared to itself is requested, then the compound object (a `Player` instance) maps `equals` to the implementation of the player.
2. $C + R_1 == C + R_1$: Same as case one, but the right-hand operator of == is a role. Here, the comparison will be done with this role's player.

3. $C + R_1 == C + R_2$: Same as case three.
4. $C == C + R_1$: We cannot modify the `equals`-method of arbitrary objects using a library approach. If the comparison of a plain player is required, the `+-Operator` needs to be applied. This will trigger the dynamic conversion using the implicit class `Player` and applies the desired comparison, as in cases one to three.

8.2.3. THE DEFINITION TABLE FOR THE PLAYS RELATIONSHIP

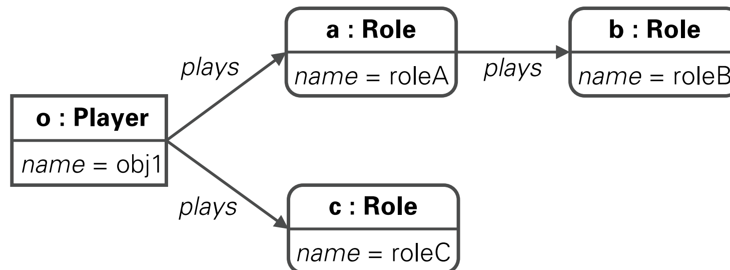


Figure 8.4.: Example of a simple role-play graph.

In *SCROLL*, a graph-based data structure is used for implementing the definition table storing the relationships between players and their roles. The role-play graph allows for easy querying of role-specific behavior that was attached to the player at some point in time. *SCROLL*'s role-play graph is defined as:

Definition 7 (*SCROLL* Role-play Graph)
A role-play graph RPG is a 4-tuple (V, E, Lab, L_Σ) with:

- V is the set of objects (player and all its attached roles),
- E is the set of edges representing the plays relationships between players and their roles,
- L_Σ is the set of type names for all objects in V, and
- $Lab : V \rightarrow L_\Sigma$ assigns each object in V its type.

As an implementation, any appropriate graph library can be used. For *SCROLL*, Guava's graph data structure [Google, 2016] was chosen as underlying graph library already providing the necessary graph-theoretic objects like edge- and node-types as well as simple algorithms for traversing the graph. *SCROLL* makes it easy to plug-in any other convenient library, e.g., for easy scaling or distribution. Additionally, access to roles is cached speeding up the querying for the appropriate structure and behavior hidden in a role. The graph traversals introduced in Sect. 5 are used and mapped directly to Scala functions as described already in Sect. 9.3.3 and with Table 9.13.

Consider the example provided in Fig. 8.4. A player type is instantiated (`o`) and plays the role type instances with the property name `roleA`, `roleB` (as deep role), and `roleC`. Hence, the *SCROLL* role-play graph is defined as:

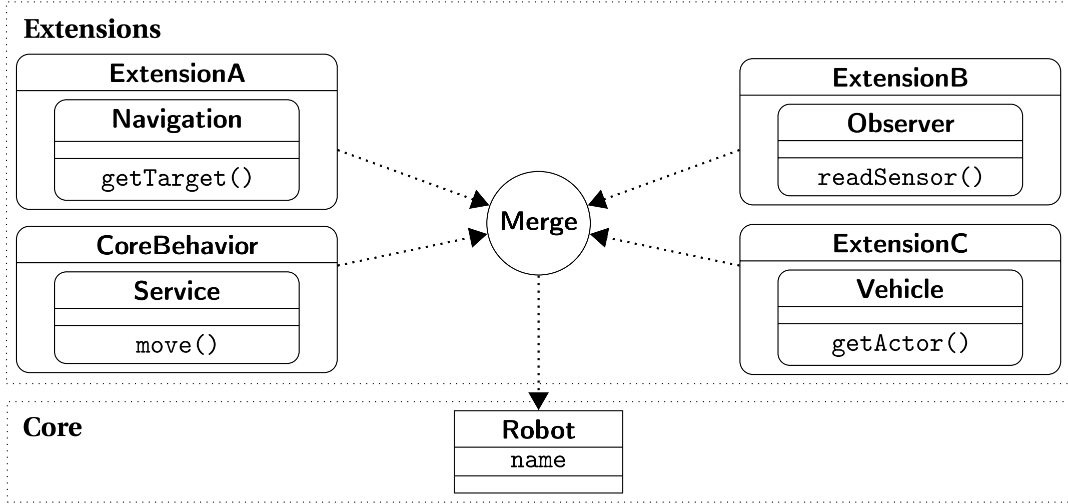


Figure 8.5.: Class `Robot` is constructed (dotted arrows) from different roles and acquires the contained behavior.

Definition 8 (Example *SCROLL* Role-play Graph)
The example role-play graph $\text{RPG}_{\text{example}}$ is a 4-tuple (V, E, Lab, L_Σ) with:

- $V := \{\text{o, a, b, c}\}$,
- $E := \{(o, a),\ (a, b),\ (o, c)\}$,
- $L_\Sigma := \{\text{Player, Role}\}$, and
- $Lab : V \rightarrow L_\Sigma := \{\text{o} \rightarrow \text{Player, a} \rightarrow \text{Role, b} \rightarrow \text{Role, c} \rightarrow \text{Role}\}$.

If i is the vertex representing the object and

$$f : \hat{\mathcal{P}}(V) \rightarrow \hat{\mathcal{P}}(S),$$

where

$$f(i,\ \text{name}) = \epsilon(\mathcal{V}_{in}(\mathcal{E}_{lab+}(\mathcal{E}_{out}(i),\ \text{plays})),\ \text{name}), \text{or more clearly as}$$

$$f(i,\ \text{name}) = (\epsilon^{\text{name}} \circ \mathcal{V}_{in} \circ \mathcal{E}^{\text{plays}}_{lab+} \circ \mathcal{E}_{out})(i,\ \text{name}),$$

then $f(i,\ \text{name})$ will return the property `name` of the roles that the object is playing. This function f traverses to the outgoing edges of vertex i representing the role-playing object, then filters those edges with the label *plays*, then traverses to the incoming (i.e., source) vertices on those plays-labeled edges. Finally, of those vertices, it returns their name property. Applying f with the player `o` and `name` now delivers $f(\text{o},\ \text{name}) = \{\text{roleA, roleB, roleC}\}$ for the example role-play graph $\text{RPG}_{\text{example}}$, as expected.

8.3. THE USAGE LAYER

This section explains the basic usage of *SCROLL* for the pure embedding of roles. We start with a brief introduction how one can use roles by example (see Fig. 8.5). A standard Scala case class (`Robot`) should be augmented with new behavior encapsulated in three different classes as extensions (`ExtensionA`, `ExtensionB` and `ExtensionC`). Each of them provides a new aspect of the robot via functions, such as, finding a target to move to, or observing sensor values, attached to case classes. This allows for a high degree of separation of concerns with multiple hierarchically structured compartments. The

core behavior (with case class `Service`) aggregates all the provided functionality without having to worry about which role delivers which service.

We now step-wise construct the example. First, only the `name` attribute of the robot should be printed. This naive solution, non-surprisingly, fails during compilation because `name()` (Line 5 in Listing 8.4) is not available at instances of `Service`.

```
1  case class Robot(name: String)
2
3  case class Service() {
4    def move() {
5      val name: String = this name()
6      info("My name is: " + name)
7    }
8  }
```

Listing 8.4.: A naive solution for the robot example. It fails during compilation because `name()` (Line 5) is not available at instances of `Service`.

To solve this problem of adding behavior dynamically, we now apply the most basic concepts of the *SCROLL* DSL, namely a `Compartment` (Line 3 in Listing 8.5), the +-operator (Line 7 in Listing 8.5), and the `play` API call (Line 12 in Listing 8.5). The `Compartment` trait (further explained in Sect. 9.2.1) exposes *SCROLL*'s basic API to the current class, allowing the programmer to use the +-operator, and the `play` method. Because any given object should be allowed to play roles, we cannot assume that this object actually provides the +-operator. Thus, Scala's implicit conversion [Odersky et al., 2008] is used to wrap the player into an equivalent compound object exposing the required API as mentioned above. By calling the +-operator, applying implicit lifting, the user is able to forward arbitrary calls to some roles he assumes should be available on the player without worrying about their actual location. Calling `play` adds a play relationship between a player (instances of `Robot`) and a role instance (instances of `Service`), finally enabling the call to `name()` (Line 7 in Listing 8.5).

```
1  case class Robot(name: String)
2
3  object CoreBehavior extends Compartment {
4
5    case class Service() {
6      def move() {
7        val name: String = +this name()
8        info("My name is: " + name)
9      }
10   }
11
12   Robot("Pete") play Service()
13 }
```

Listing 8.5.: A new solution for the robot example using the basic *SCROLL* API.

As a final step, for better separation of concerns, new functionality from roles is now grouped into extensions represented by individual compartments, e.g., with the compartment `ExtensionA` (Line 15 in Listing 8.6).

```
case class Robot(name: String)

object CoreBehavior extends Compartment {

  case class Service() {
    def move() {
      val name: String = +this name()
      val target: String = +this getTarget()
      info(s"$name moves to $target.")
    }
  }

}

object ExtensionA extends Compartment {

  case class Navigation() {
    def getTarget = "kitchen"
  }

}

new Compartment {
  val robot = Robot("Pete") play Service() play Navigation()
  ExtensionA partOf CoreBehavior partOf this
  robot move()
}
```

Listing 8.6.: The third solution for the robot example using the more advanced *SCROLL* API.

The role-playing graph, holding the relationships between role-playing objects (e.g., instances of `Robot`) and their roles (e.g., instances of `Service`, and `Navigation`), is defined compartment-wise. Hence, in the anonymously instantiated compartment at Line 23 in Listing 8.6, making the robot actually move, those individual role-playing graphs are merged into a new one, spanning now multiple compartment instances (the operator `partOf` is explained in Sect. 9.2.1). With that, the role-playing relationships defined in the anonymously instantiated compartment are now a part of those within `CoreBehavior`, and `ExtensionA`, respectively. Hence, all the requested behavior (i.e., `name()`, `getTarget()`, and `move()`) is available. The full example can be found in Listing 8.7.

In sum, with targeting the end-user of *SCROLL* directly, the usage layer is tailored for the actual instantiating and use of role-playing objects. Players and roles are instantiated from standard Scala classes or case classes, compartments from traits. In-depth explanations for the individual usage via instantiating from components out of the metaobject protocol layer, is presented in Sect. 9.2.

```
case class Robot(name: String)

object CoreBehavior extends Compartment {

  case class Service() {
    def move() {
      val name: String = +this name()
      val target: String = +this getTarget()
      val sensorValue: Int = +this readSensor()
      val actor: String = +this getActor()
      info(s"$name moves to $target with $actor and sensor value of $sensorValue.")
    }
  }

}

object ExtensionA extends Compartment {

  case class Navigation() {
    def getTarget = "kitchen"
  }

}

object ExtensionB extends Compartment {

  case class Observer() {
    def readSensor = 100
  }

}

object ExtensionC extends Compartment {

  case class Vehicle() {
    def getActor = "wheels"
  }

}
```

Listing 8.7.: The `RobotExample` model source code.

```
new Compartment {
  val myRobot = Robot("Pete") play Service() play Navigation() play Observer()
      play Vehicle()

  ExtensionC partOf ExtensionB partOf ExtensionA partOf CoreBehavior partOf this

  myRobot move()
}
```

Listing 8.8.: The `RobotExample` instance source code.

```
Pete moves to kitchen with wheels and sensor value of 100.
```

Listing 8.9.: The `RobotExample` console output.

Figure 8.6.: The robot is constructed from multiple roles dynamically at runtime.

9

THE METAOBJECT PROTOCOL OF *SCROLL*

A *metaclass* is a class containing a structural or behavioral description of its instances, classes. A *metaobject protocol* (MOP) is the protocol followed by the set of metaclasses of a language. The protocol followed by the *SCROLL* metaclasses provides the behavior of the reference implementation *SCROLL* and is called the *SCROLL* MOP [Kiczales et al., 1991; Foreman and Danforth, 1998]. This metaobject protocol is mapped directly to a set of interfaces of a library (an API), the *SCROLL* library.

metaclass
metaobject protocol

The *SCROLL* library is split into an implementation of the main metaclass `Compartment` and some utility metaclasses as Scala traits. The first concept encapsulates the semantics of role evolution, while the latter ones handle related role-specific features. With the *SCROLL* MOP, the behavior of the metaclasses (like, method dispatch and instantiation) is defined. Within the open implementation principle, slices of the library can be varied at runtime. Dispatch is implemented by filtering and then calling the individual role. Communication between the player and its roles is accomplished via consultation specific to the enclosing compartment instance. In the following, we provide an in-depth description of the *SCROLL* MOP metaclasses for those library developers who want to adapt or transfer the *SCROLL* MOP to their specific research area.

9.1. THE CONFIGURATION LAYER

This layer is targeted to both end-users and library developers. All role-specific features of a compound object are mixed into a Scala object by inheriting from the trait `Compartment`, and, in turn, from its supporting traits (as listed below). Two additional traits implement the semantics for the binding of roles (`RolePlayingAutomaton`) and their definition table (`RoleGraph`). Those traits implement the full interface (API) of the *SCROLL* MOP and are configurable at runtime through polymorphic variation of their concrete instances. Altering their default behavior is viable via subclassing and polymorphism. The API available for configuring this layer can be found in Sect. 9.2 as follows:

Trait `Compartment` This trait represents a reified collaboration with a limited number of participating roles and a fixed scope. Mixing in the `Compartment` trait exposes *SCROLL*'s basic API to the current class. Contained classes or case classes (roles and other compartments) can be seen as containers for new behavior and state that should be attached later on (Sect. 9.2.1).

Implicit class `Player` This class wraps the player and its roles with a compound object. Additionally, the role-playing object semantics is exposed (Sect. 9.2.2).

Class `DispatchQuery` The selection of the correct context-dependent behavior from all available roles is supported by *SCROLL* with function composition and Scala's pattern matching making use of an explicit dispatch description. The `DispatchQuery` command object is passed to the actual method invocation as implicit argument. The given selection functions are applied while traversing the role-play graph holding the relationships between players and their role instances. For that, `DispatchQuery` provides some static dispatch functions and a dispatch query creation API (Sect. 9.2.3).

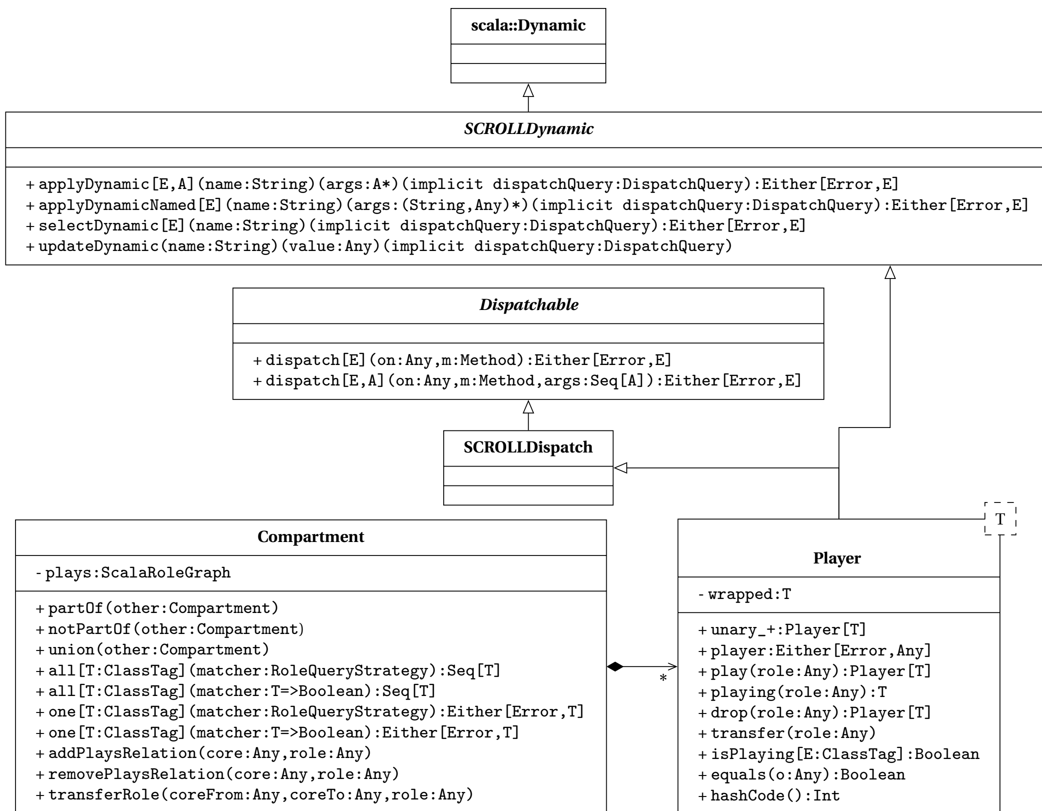


Figure 9.1.: The general API design of the *SCROLL* library.

Trait `RoleGraph` This trait implements the definition table for storing the relationships between players and their roles. It allows for querying role-specific behavior that was attached to the player (Sect. 9.2.4).

Trait `QueryStrategies` This API allows for writing queries looking up the content of an attribute of the certain compound object or the return value of one of its functions (Sect. 9.2.5).

Trait `RoleConstraints` This API allows for adding and checking role constraints as in Riehle [2000] within a specific compartment instance (Sect. 9.2.6).

Trait `RoleRestrictions` This API allows for adding and checking role restrictions (in the sense of structural typing) within a specific compartment instance (Sect. 9.2.7).

Trait `Relationships` This trait allows for adding role relationships with multiplicities to a compartment instance. Additionally, these relationship multiplicity constraints can be checked. Furthermore, querying for concrete instances of the association ends for each defined relationship is possible (Sect. 9.2.8).

Trait `RoleGroups` This API allows for the creation and the runtime check of role groups (see Sect. 3.5) in specific compartment instances (Sect. 9.2.9).

Trait `RolePlayingAutomaton` Using this trait allows for implementing compartment-specific role-playing automatons to specify the role life-cycle with e.g., the binding, unbinding or transfer of role instances between objects as introduced in Sect. 3.6. Predefined event types for this automaton are available in its companion object (see Appendix B). It is based on the implementation of finite state machines from the *Akka* framework [Lightbend Inc., 2016a] (Sect. 9.2.10).

9.2. THE METAOBJECT PROTOCOL LAYER

The section reveals one of the major contributions of the thesis, because it demonstrates that the semantics of role-based programming can be hidden by the interception of method calls, i.e., that *SCROLL* is a method-call interception based DSL. The MOP layer contains the actual implementation of the metaclass `Compartment`, its helper traits, and helper classes. Especially the dynamic dispatch semantics within the metaclass `DispatchQuery` can be investigated and adapted by library developers. An overview of the general API design can be found in Fig. 9.1. Additionally, the following sections provide a more detailed description of the components found in this layer together with examples. All methods not explicitly defined with the return type of `Either[Error, _]` will fail (e.g., if the desired role-playing relationship cannot be found) with an appropriate runtime exception.

9.2.1. THE METACLASS COMPARTMENT

Fully qualified class name
`scroll.internal.Compartment`

Full source code
See Listing D.6 on page 189

API
```
protected val plays: ScalaRoleGraph
def partOf(other: Compartment)
def union(other: Compartment)
def all[T: ClassTag](matcher: RoleQueryStrategy): Seq[T]
def all[T: ClassTag](matcher: T => Boolean): Seq[T]
def one[T: ClassTag](matcher: RoleQueryStrategy): Either[Error, T]
def one[T: ClassTag](matcher: T => Boolean): Either[Error, T]
def addPlaysRelation(core: Any, role: Any)
def removePlaysRelation(core: Any, role: Any)
def transferRole(coreFrom: Any, coreTo: Any, role: Any)
```

Table 9.1.: Overview for `Compartment`.

This section explains why *SCROLL* supports context-oriented programming, in the style of CROM [Kühn et al., 2014]. Compartments provide structured contexts with a fixed set of relationships between their included roles in the *SCROLL* MOP. Mixing in the `Compartment` trait exposes *SCROLL*'s basic API to the current class. Contained classes or case classes can be seen as containers for new behavior and state that should be attached later on. An example usage and configuration of the `Compartment` trait looks like this (Listing 9.1):

```
1 // create a player instance
2 val player = new Player()
3 new Compartment { // create an anonymous Compartment instance
4   // define two roles
5   class RoleA; class RoleB
6   // create the play-relationships
7   player play new RoleA(); player play new RoleB()
8   // call some behavior ...
9 }
```

Listing 9.1.: `Compartment` usage example.

An overview of the API can be found in Table 9.1. The following provides a more detailed description together with examples:

`protected val plays: ScalaRoleGraph`
: This attribute holds the role-play graph. A more in-depth explanation can be found in Sect. 8.2.3 and Sect. 9.2.4. This field may be overridden by custom subclasses of the `Compartment` to define new behavior, e.g., like this:

```
1 new Compartment {
2   // use a cached variant of the role-play graph from now on,
      additionally disable all checks for cyclic role-playing
      relationships
3   override val plays = new CachedScalaRoleGraph(checkCycles = false)
4   /* ... */
5 }
```

`def partOf(other: Compartment)`
: This function allows for the definition of an is-part-of relation between compartments. For instance, in the case of nested compartments, one is able to merge their individual role graphs.

`def union(other: Compartment)`
: This function declares a bidirectional is-part-of relation between compartments, i.e., it is semantically identical to:

```
1 this.partOf(other)
2 other.partOf(this)
```

`def all[T: ClassTag](matcher: RoleQueryStrategy): Seq[T]`
: This function queries the role-play graph for all player instances that conform to the given matcher. `T` is the type of the player instance to query for. It returns all player instances that conform to the given matcher as sequence. Example usages might be:

```
1 // query for all players with type Robot in the surrounding compartment
    instance
2 val allRobots = all[Robot]()
3 // same query, but restrict the search to robots with the given name
4 val someRobots = all[Robot]("name" ==# "Kuka")
```

`def all[T: ClassTag](matcher: T => Boolean): Seq[T]`
: This queries the role-play graph for all player instances that do conform to the given function. `T` is the type of the player instance to query for. It returns all player instances as a sequence that conform to the given function. Example usages might be:

```
1 // query for all players with type Robot in the surrounding compartment
    instance
2 val allRobots = all[Robot]()
3 // same query, but restrict the search to robots with the given name
4 val someRobots = all[Robot](_ match {
5   case r: Robot if r.name == "Kuka" => true
6   case _ => false
7 })
```

`def one[T: ClassTag](matcher: RoleQueryStrategy): Either[Error, T]`
: This function queries the role-play graph for all player instances that do conform to the given matcher and return the first found. `T` is the type of the player instance to query for. It returns the first player instance that does conform to the given matcher or an appropriate error. Usage is the same as shown above for `all`.

`def one[T: ClassTag](matcher: T => Boolean): Either[Error, T]`

This function queries the role-play graph for all player instances that conform to the given function and returns the first found. `T` is the type of the player instance to query for. Its usage is the same as shown above for `all`.

`def addPlaysRelation(core: Any, role: Any)`

This function adds a play relationship between the given player and a role. It is intended to be used from the outside of a concrete compartment instance, e.g., like in the following:

```
// create a player instance
val player = new KukaRobot()
// define a new compartment type
class RobotCompartment extends Compartment {
  // define two roles
  class Autonomous { /* ... */ }
  class HumanFriendly { /* ... */ }
}

// create a new instance of that compartment
val robotCompartment = new RobotCompartment()

// create the play-relationship
robotCompartment.addPlaysRelation(player, new robotCompartment.Autonomous())

// call some behavior
/* ... */
}
```

`def removePlaysRelation(core: Any, role: Any)`

This function removes a play relationship between the given player and a role and is intended to be used from the outside of a concrete compartment instance, analogous to the example for `addPlaysRelation` above.

`def transferRole(coreFrom: Any, coreTo: Any, role: Any)`

This function transfers a role instance from one player to another and is semantically identical to:

```
removePlaysRelation(coreFrom, role)
addPlaysRelation(coreTo, role)
```

9.2.2. THE METACLASS PLAYER

Fully qualified class name `scroll.internal.Compartment#Player` (as inner class)
Full source code See Listing D.6 on page 189

API
`val wrapped: T`
`def unary_+: Player[T]`
`def player(implicit dispatchQuery: DispatchQuery): Either[TypeError, Any]`
`def play(role: Any): Player[T]`
`def playing(role: Any): T`
`def drop(role: Any): Player[T]`
`def transfer(role: Any)`
`def isPlaying[E: ClassTag]: Boolean`

Table 9.2.: Overview for `Player`.

This section presents how *SCROLL* supports role-based programming with contexts. The class `Player` encapsulates the player and its currently played roles. Hence, a `Player` instance represents a compound object. Additionally, the role API is exposed. An overview of the API can be found in Table 9.2. The following provides a more detailed description together with examples:

`val wrapped: T`
: This attribute stores the player or role encapsulated in the compound object that is represented with an instance of `Player`.

`def unary_+: Player[T]`
: In Scala, method calls can be written as prefix operators. `+this` is equivalent to `this.+()`. Because any given object should be able to play roles, we cannot assume that this object actually provides the +-operator. Thus, Scala's implicit conversion [Odersky et al., 2008] is used to wrap the player with a compound object that exposes the required API. In summary, by calling the +-operator, and applying implicit lifting, the developer is able to forward arbitrary calls using consultation to some roles that he assumes should be available on the player. An example might be:

```
1  // create a player instance
2  val player = new Player()
3
4  // create an anonymous Compartment instance
5  new Compartment {
6
7    // define a role
8    class RoleA {
9      def behavior(): Unit = { /* ... */ }
10   }
11
12   // create the play-relationship
13   player play new RoleA()
14
15   // call the behavior using the +-operator
16   +player behavior()
17 }
```

`def player(implicit dispatchQuery: DispatchQuery): Either[TypeError, Any]`
: This function returns the player instance (the role-playing object). The parameter `dispatchQuery` provides means for filtering and sorting the result if multiple role-playing objects exist like explained in Sect. 9.3.3.

`def play(role: Any): Player[T]`
: This function adds a play relationship between a player and a role instance. Examples can be found in the listings above. This function returns the resulting compound object that is represented as new `Player` instance.

`def playing(role: Any): T`
: This function is the same as `play`, but returns the innermost player instance.

`def drop(role: Any): Player[T]`
: This function removes the play relationship between a player and a role instance. It returns the compound object represented as `Player` instance.

`def transfer(role: Any)`
: This function transfers a role from one player to another. An example might be:

```
1  val playerA = new Player()
2  val playerB = new Player()
3
4  new Compartment {
5    class RoleA
6    val roleA = new RoleA()
7    playerA play roleA
8    // transfer the role
9    playerA transfer roleA to playerB
10 }
```

`def isPlaying[E: ClassTag]: Boolean`
: This function checks if a player is playing a role of the given type or has the given role attached to itself. An example might be:

```
1  val playerA = new Player()
2  val playerB = new Player()
3
4  new Compartment {
5    class RoleA
6    val roleA = new RoleA()
7    playerA play roleA
8    playerA transfer roleA to playerB
9    // after the transfer, playerA should not be playing an instance of
       RoleA anymore, but instead playerB should be now
10   assertFalse(+playerA isPlaying[RoleA])
11   assertTrue(+playerB isPlaying[RoleA])
12 }
```

9.2.3. THE METACLASS DISPATCHQUERY

Fully qualified class name
`scroll.internal.support.DispatchQuery`

Full source code
See Listing D.7 on page 198

API
```
val identity: Boolean = false
val swap: Boolean = true
val reverse: PartialFunction[(Any, Any), Boolean]
val anything: Any => Boolean
val nothing: Any => Boolean
def From(f: Any => Boolean) To(t: Any => Boolean) Through(th: Any =>
    Boolean) Bypassing(b: Any => Boolean): DispatchQuery
def Bypassing(b: Any => Boolean): DispatchQuery
def empty: DispatchQuery
def sortedWith(f: PartialFunction[(Any, Any), Boolean]): DispatchQuery
```

Table 9.3.: Overview for `DispatchQuery`.

Sometimes it is ambiguous or context-dependent which role should be selected for answering a call. The developer should be able to specify the desired selection. *SCROLL* supports this with function composition and Scala's pattern matching making use of an explicit dispatch description which is passed to the actual method invocation as implicit argument. The given selection functions are applied while traversing the role-play graph holding the relationships between all players and their currently played roles. For this, the `DispatchQuery` provides dispatch functions and a dispatch query creation API. This results in a dispatch query (i.e., a higher-order adjacency) that is a composition of all dispatch functions over the given set of nodes (roles in the role-play graph) utilizing the functions `filter` and `sortedWith`. All provided filters must be side-effect free. A more in-depth explanation can be found in Sect. 9.3.3. An overview of the API can be found in Table 9.3. The following provides a more detailed description together with examples:

`val identity: Boolean = false`
: This constant is meant to be used in `sortedWith` to state that no sorting between the objects in comparison should happen.

`val swap: Boolean = true`
: This constant is meant to be used in `sortedWith` to state that always swapping between the objects in comparison should happen.

`val reverse: PartialFunction[(Any, Any), Boolean]`
: This partial function is meant to be used in `sortedWith` to state that a reversing of the collection of resulting nodes (i.e., role instances) should happen.

`val anything: Any => Boolean`
: This function will always return `true`. The resulting dispatch query will select all roles encapsulated in the compound object.

`val nothing: Any => Boolean`
: This function will always return `false`. The resulting dispatch query will select no role encapsulated in the compound object.

`def From(f: Any => Boolean) To(t: Any => Boolean) Through(th: Any => Boolean) Bypassing(b: Any => Boolean): DispatchQuery`

This function realizes the path filter from Chapter 5. It constructs a new dispatch query from the given selection functions `f` (selecting the set of nodes qualifying for f), `t` (compute all reachable roles from the set of nodes qualifying for the `From` clause), `th` (specifying the roles to keep), and `b` (specifying the roles to skip). An example is:

```
// create a player instance
val player = new Player()

// create an anonymous Compartment instance
new Compartment {
  // define two roles with the same behavior but different implementations
  class RoleA {
    def behavior(): Unit = {
      doA()
    }
  }
  class RoleB {
    def behavior(): Unit = {
      doB()
    }
  }

  // create the play-relationships
  player play new RoleA()
  player play new RoleB()

  // we want to select the implementation of RoleB
  implicit val dd =
    From(_.isInstanceOf[Player]).
    To(anything).
    Through(anything).
    Bypassing(_.isInstanceOf[RoleA])

  // call the actual behavior
  +player behavior()
}
```

`def Bypassing(b: Any => Boolean): DispatchQuery`

Short-hand factory function for constructing a new, empty `DispatchQuery` selecting all role instances encapsulated in the compound object, but skipping those matching the given function `b`.

`def empty: DispatchQuery`

Short-hand factory function for constructing a new, empty `DispatchQuery` selecting all role instances encapsulated in the compound object and skipping nothing.

`def sortedWith(f: PartialFunction[(Any, Any), Boolean]): DispatchQuery`

This factory function constructs a new `DispatchQuery` and sorts the resulting collection of role instances based on a configurable sorting function. This can be mixed with the factory methods described above. An example is:

```
// This will always reverse the collection of role instances gathered in the compound object:
implicit var dd .sortedWith(reverse)

// Type-based comparison: reverse the order of the role instances encapsulated in the compound object if they are of type SomeRoleB and SomeRoleC:
dd .sortedWith {
  case (_: SomeRoleB, _: SomeRoleC) => swap
}

// Filtering and type-based comparison: reverse the order of the role instances encapsulated in the compound object if they are of type SomeRoleB and SomeRoleC, but filter out all instances of SomeRoleA first:
dd = Bypassing(_.isInstanceOf[SomeRoleA]).sortedWith {
  case (_: SomeRoleB, _: SomeRoleC) => swap
}
```

9.2.4. THE METACLASS ROLEGRAPH

Fully qualified class name
scroll.internal.graph.RoleGraph
Full source code
See Listing D.14 on page 217
API

```
def merge(other: RoleGraph): Unit
def detach(other: RoleGraph): Unit
def addBinding[P <: AnyRef : ClassTag, R <: AnyRef : ClassTag](player: P,
    role: R): Unit
def removeBinding[P <: AnyRef : ClassTag, R <: AnyRef : ClassTag](player:
    P, role: R): Unit
def removePlayer[P <: AnyRef : ClassTag](player: P): Unit
def allPlayers: Seq[Any]
def getRoles(player: Any)(implicit dispatchQuery: DispatchQuery ): Set[Any]
def containsPlayer(player: Any): Boolean
def getPredecessors(player: Any)(implicit dispatchQuery: DispatchQuery ):
    Seq[Any]
```

Table 9.4.: Overview for RoleGraph.

This section explains the role-play graph, the basis for the 4-dimensional dispatch in *SCROLL*. In *SCROLL*, the role-play graph allows for easy querying of role-specific behavior that was attached to the player at some point in time earlier on. A more in-depth explanation can be found in Sect. 8.2.3. An overview of the API can be found in Table 9.4. The following provides a more detailed description:

`def merge(other: RoleGraph): Unit`
: This function adds all plays relationships from other to this role-play graph instance. This allows for creating hierarchically nested compartments and plays relationships spanning over multiple compartment instances.

`def detach(other: RoleGraph): Unit`
: This function removes all plays relationships from other in this role-play graph instance.

`def addBinding[P <: AnyRef : ClassTag, R <: AnyRef : ClassTag](player: P, role: R): Unit`
: This function adds a plays relationship between the given player and a role instance.

`def removeBinding[P <: AnyRef : ClassTag, R <: AnyRef : ClassTag](player: P, role: R): Unit`
: This function removes a plays relationship between the given player and a role instance.

`def removePlayer[P <: AnyRef : ClassTag](player: P): Unit`
: This function removes a player from the role-play graph completely. Plays relationships between the given player and its role instances are removed as well.

`def allPlayers: Seq[Any]`
: This function returns a sequence of all players (all players, and all role instances) stored in this graph instance.

`def getRoles(player: Any)(implicit dispatchQuery: DispatchQuery ): Set[Any]`

This function returns a set of all roles attached to the given player. The given `dispatchQuery` is applied while traversing the role-play graph.

`def containsPlayer(player: Any): Boolean`

This function checks if the role-play graph contains the given player and returns `true` if it does, `false` otherwise.

`def getPredecessors(player: Any)(implicit dispatchQuery: DispatchQuery ): Seq[Any]`

This function returns a sequence of all players the given compound object has. The given `dispatchQuery` is applied while traversing the role-play graph.

9.2.5. THE METACLASS QUERYSTRATEGIES

Fully qualified class name
scroll.internal.support.QueryStrategies
Full source code
See Listing D.8 on page 201

API
 `def ==#[T](value: T): WithProperty[T]`
 `def ==>[T](value: T): WithResult[T]`

Table 9.5.: Overview for `QueryStrategies`.

Using this API allows for writing queries looking for the content of an attribute of a compound object or the return value of one of its functions. An overview of the API can be found in Table 9.5. The following provides a more detailed description together with examples:

`def ==#[T](value: T): WithProperty[T]`
This function returns the value of the queried attribute (invoking this attribute reflectively). An example usage might be:

```
1 // query for all players with type Robot, but restrict the search to
    robots with the given name
2 val someRobots = all[Robot]("name" ==# "Kuka")
```

`def ==>[T](value: T): WithResult[T]`
This function returns the result of the queried function (invoking this function reflectively). An example usage might be:

```
1 // query for all players with type Robot, but restrict the search to
    robots that can drive
2 val someRobots = all[Robot]("canDrive" ==> true)
```

9.2.6. THE METACLASS ROLECONSTRAINTS

Fully qualified class name
`scroll.internal.support.RoleConstraints`
Full source code
See Listing D.10 on page 204
API

`def RoleImplication[A : ClassTag , B : ClassTag](): Unit`
`def RoleEquivalence[A : ClassTag , B : ClassTag](): Unit`
`def RoleProhibition[A : ClassTag , B : ClassTag](): Unit`
`def RoleConstraintsChecked(func: => Unit): Unit`

Table 9.6.: Overview for `RoleConstraints`.

This section shows how *SCROLL* supports constraint models for 4-dimensional dispatch. Using this API allows for adding and checking the role constraints specified in Riehle [2000] within a specific compartment instance. Those constraints are explained in more detail in Sect. 3.5. An overview of the API can be found in Table 9.6. The following provides a more detailed description together with examples:

`def RoleImplication[A : ClassTag , B : ClassTag](): Unit`
: This function adds a role implication constraint between the given role types. If a player plays an instance of role type `A`, it also has to play an instance of role type `B`.

`def RoleEquivalence[A : ClassTag , B : ClassTag](): Unit`
: This function adds a role equivalent constraint between the given role types. If a player plays an instance of role type `A`, it also has to play an instance of role type `A` and visa versa.

`def RoleProhibition[A : ClassTag , B : ClassTag](): Unit`
: This function adds a role prohibition constraint between the given role types. If a player plays an instance of role type `A`, it is not allowed to play any instance of `B` as well.

`def RoleConstraintsChecked(func: => Unit): Unit`
: This function allows for checking all available role constraints for all compound objects after the given function was executed in the currently active compartment. A RuntimeException will be thrown if a role constraint is violated. An example is:

```
1  val player = new Player()
2  new Compartment {
3    class RoleA; class RoleB; class RoleC
4    // add a role implication constraint
5    RoleImplication[RoleA , RoleB]()
6    // and check them
7    RoleConstraintsChecked { player play roleA play roleB }
8    // this will throw a RuntimeException because it violates the given
       role implication constraint from above
9    RoleConstraintsChecked { player drop roleB }
10 }
```

9.2.7. THE METACLASS ROLERESTRICTIONS

Fully qualified class name
scroll.internal.support.RoleRestrictions
Full source code See Listing D.12 on page 212

API
`def RoleRestriction[A : ClassTag , B : ClassTag](): Unit`
`def ReplaceRoleRestriction[A : ClassTag , B : ClassTag](): Unit`
`def validate[R : ClassTag](player: Any , role: R): Unit`

Table 9.7.: Overview for RoleRestrictions.

This section explains how *SCROLL* supports restrictions on role groups, as challenged from CROM [Kühn et al., 2014]. Using this API allows for adding and checking role restrictions (in the sense of structural typing) within a specific compartment instance. An overview of the API can be found in Table 9.7. The following provides a more detailed description together with examples:

`def RoleRestriction[A : ClassTag , B : ClassTag](): Unit`
: This function adds a role restriction between the given player type `A` and role type `B`. Instances of `A` are only allowed to play roles of instance of `B`. See `validate` for an example usage.

`def ReplaceRoleRestriction[A : ClassTag , B : ClassTag](): Unit`
: This function replaces a role restriction for the given player type `A` with a restriction to role type `B`. See `validate` for an example.

`def validate[R : ClassTag](player: Any , role: R): Unit`
: Every operation manipulating the role-play graph in a concrete compartment instance (e.g., `play`, `drop`, or `transfer`) will call this validation method implicitly. If any of the added restrictions is violated, a `RuntimeException` will be thrown. See the following example for the general usage:

```
1  // create a player instance
2  val player = new Player()
3  // create an anonymous Compartment instance
4  new Compartment {
5    // define some roles
6    class RoleA
7    class RoleB
8    class RoleC
9    // add role restrictions. Here, instances of Player should only be
       allowed to play instances of RoleA and RoleB
10   RoleRestriction[Player, RoleA]
11   RoleRestriction[Player, RoleB]
12   // then, no restriction is violated
13   player play new RoleA()
14   player play new RoleB()
15   // but this will throw a RuntimeException because it violates the
       given role restrictions
16   player play new RoleC()
17 }
```

9.2.8. THE METACLASS RELATIONSHIPS

Fully qualified class name
`scroll.internal.support.Relationships`
Full source code
See Listing D.9 on page 202
API

`def left(matcher: L => Boolean = _ => true): Seq[L]`
 `def right(matcher: R => Boolean = _ => true): Seq[R]`
 `def apply(name: String) from[L: ClassTag](leftMul: Multiplicity) to[R: ClassTag](rightMul: Multiplicity): Relationship[L, R]`

Table 9.8.: Overview for `Relationships`.

SCROLL supports binary relationships with sides *A* and *B*. With the addition of relationships, it becomes clear, why *SCROLL* supports all aspects of role-based languages, i.e., their structural, behavioral, relational, and contextual nature (see Sect. 3.4). This trait allows for adding and checking role relationships with arbitrary, predefined multiplicities to a compartment instance. Additionally, querying for concrete instances of the association ends for each defined relationship is possible. An overview of this API can be found in Table 9.8. The following provides a more detailed description together with examples:

`def left(matcher: L => Boolean = _ => true): Seq[L]`
: This function returns all instances of side *A* of the relationship with regard to the provided matching function. The multiplicity is checked additionally, throwing a `RuntimeException` if more objects than specified are available within the role-play graph.

`def right(matcher: R => Boolean = _ => true): Seq[R]`
: This function returns all instances of side *B* of the relationship with regard to the provided matching function. The multiplicity is checked additionally, throwing a `RuntimeException` if more objects than specified are available within the role-play graph.

`def apply(name: String) from[L: ClassTag](leftMul: Multiplicity) to[R: ClassTag](rightMul: Multiplicity): Relationship[L, R]`
: This factory method creates a new relationship object with the given multiplicities for side *A* and *B* and their types. An example is:

```
1  val player = new Player()
2  new Compartment {
3    class RoleA; class RoleB
4    val roleA = new RoleA(); val roleB1 = new RoleB()
5    player play roleA play roleB1
6    // create a new 1-to-1 relationship instance
7    val rel1 = Relationship("rel1").from[RoleA](1).to[RoleB](1)
8    assert(rel1.left() == Seq(roleA))
9    assert(rel1.right() == Seq(roleB1))
10   // and one additional binding
11   val roleB2 = new RoleB()
12   player play roleB2
13   // together with a new 1-to-many relationship
14   val rel2 = Relationship("rel2").from[RoleA](1).to[RoleB](*)
15   assert(rel2.right() == Seq(roleB1, roleB2))
16 }
```

9.2.9. THE METACLASS ROLEGROUPS

Fully qualified class name
scroll.internal.support.RoleGroups
Full source code
See Listing D.11 on page 207
API
def RoleGroupsChecked(func: => Unit): Unit def apply(name: String) containing(rg: RoleGroup*)(limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup def apply(name: String) containing[T1: ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup def apply(name: String) containing[T1: ClassTag, T2: ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup def apply(name: String) containing[T1: ClassTag, T2: ClassTag, T3: ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup

Table 9.9.: Overview for RoleGroups.

This section explains how *SCROLL* supports role groups, as introduced by Kühn et al. [2014]. This API allows for the creation and the runtime check of role groups in specific compartment instances. An overview of this API can be found in Table 9.9. The following provides a more detailed description together with examples:

`def RoleGroupsChecked(func: => Unit): Unit`

This function checks all available role group constraints for all player and their roles, after the given function was executed in the currently active compartment instance. This will throw a RuntimeException if a role group constraint is violated. See the following example for its usage:

```
1  val playerA = new Player()
2  val playerB = new Player()
3
4  new Compartment {
5    class RoleA
6    class RoleB
7    val roleA = new RoleA()
8    val roleB = new RoleB()
9
10   // create a new RoleGroup instance
11   val roleGroup = RoleGroup("roleGroup").containing[RoleA, RoleB](1,
       1)(2, 2)
12
13   // No exception, since this is compliant to the role group constraints:
14   RoleGroupsChecked { playerA play roleA; playerB play roleB }
15
16   // This will throw a RuntimeException, since an instance of RoleB is
       only allowed to be played exactly once:
17   RoleGroupsChecked { playerB drop roleB }
18
19   // This will throw a RuntimeException as well, since an instance of
       role type RoleA is only allowed to be played exactly once:
20   RoleGroupsChecked { playerA play roleA }
21 }
```

```
def apply(name: String) containing(rg: RoleGroup*)(limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup
```
This function creates a new `RoleGroup` instance with the given name, inner, and occurrence constraints, and hierarchical nesting into other role groups.

```
def apply(name: String) containing[T1 : ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup
```
This function creates a new `RoleGroup` instance with the given name, inner-, and occurrence constraints for type `T1`.

```
def apply(name: String) containing[T1 : ClassTag, T2 : ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup
```
This function creates a new `RoleGroup` instance with the given name, inner-, and occurrence constraints for type `T1` and `T2`.

```
def apply(name: String) containing[T1 : ClassTag, T2 : ClassTag, T3 : ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup
```
This function creates a new `RoleGroup` instance with the given name, inner-, and occurrence constraints for type `T1`, `T2`, and `T3`. For more type parameters, additional factory methods up to five types are available, as well.

9.2.10. THE METACLASS ROLEPLAYINGAUTOMATON

Fully qualified class name
`scroll.internal.rpa.RolePlayingAutomaton`
Full source code
See Listing D.13 on page 214
API

```
trait RPAState
trait RPAData
def run(): Unit
def halt(): State
def Use[T](implicit ct: ClassTag[T]) For(comp: Compartment): ActorRef
def stay(): State
def goto(nextStateName: S): State
def onTransition(transitionHandler: TransitionHandler): Unit
def when(state: S)(stateFunction: StateFunction): Unit
def whenUnhandled(stateFunction: StateFunction): Unit
```

Table 9.10.: Overview for `RolePlayingAutomaton`.

This section shows how *SCROLL* realizes the role-playing automaton for player objects. Using this trait allows for implementing compartment specific role-playing automatons to specify the role life-cycle with the binding, unbinding or transfer of role instances between objects as introduced in Sect. 3.6. Predefined event types for messaging are available in the companion object (see Appendix B). In *SCROLL*, it is based on the implementation of finite state machines from the *Akka* framework [Lightbend Inc., 2016a]. An overview of the API can be found in Table 9.10. The following provides a more detailed description:

`trait RPAState`
: This trait should be extended for defining states.

`trait RPAData`
: This trait should be extended for defining messages the automaton should react on during its life-cycle.

`def run(): Unit`
: This function starts the automaton, i.e., starts the background actor system from *Akka* [Lightbend Inc., 2016a] associated with it.

`def halt(): State`
: This function stops the automaton, i.e., terminates the background actor system associated with it.

`def Use[T](implicit ct: ClassTag[T]) For(comp: Compartment): ActorRef`
: This function associates the automaton with the background actor system.

`def stay(): State`
: This function produces an empty transition descriptor. This state function should be returned when no state change is desired.

`def goto(nextStateName: S): State`
: This function triggers a transition to another state.

`def onTransition(transitionHandler: TransitionHandler): Unit`
: This function sets the handler, which is called upon each state transition. Multiple handlers may be installed.

`def when(state: S)(stateFunction: StateFunction): Unit`
: This function inserts a new function at the end of the processing chain for the given state.

`def whenUnhandled(stateFunction: StateFunction): Unit`
: This function sets a handler, which is called upon reception of otherwise unhandled messages.

See the following example (Listing 9.2) for its usage:

```
1  val player = new Player()
2  class ACompartment extends Compartment {
3    val roleA = new RoleA(); val roleB = new RoleB(); val roleC = new RoleC()
4    // define a role-playing automaton for this compartment
5    class MyRPA extends RolePlayingAutomaton {
6      // with some states
7      private case object StateA extends RPAState
8      private case object StateB extends RPAState
9      private case object StateC extends RPAState
10     // implement the actual transition logic
11     when(Start) {
12        case Event(BindRole, _) => goto(StateA)
13     }
14     when(StateA) {
15       case Event(BindRole, _) => goto(StateB)
16     }
17     when(StateB) {
18        case Event(BindRole, _) => goto(StateC)
19     }
20     when(StateC) {
21        case Event(Terminate, _) => goto(Stop)
22     }
23     onTransition {
24        case Start -> StateA => player play roleA; self ! BindRole
25        case StateA -> StateB => player play roleB; self ! BindRole
26        case StateB -> StateC => player play roleC; self ! Terminate
27     }
28     run()
29   }
30   // bind the automaton to the currently active compartment instance and run it
31   (Use[MyRPA] For this) ! BindRole
32 }
33 new ACompartment() {
34   // after running the automaton the following bindings should be applied
35   (+player).isPlaying[RoleA] shouldBe true
36   (+player).isPlaying[RoleB] shouldBe true
37   (+player).isPlaying[RoleC] shouldBe true
38 }
```

Listing 9.2.: Example for using the `RolePlayingAutomaton`.

It is particularly interesting that *SCROLL*, for the use of the role-play automaton, does not require to extend the syntax of Scala. Everything stays in the framework of a method-call interception DSL.

9.3. THE SPECIFICATION LAYER

To handle the actual dispatching on the compound object, this layer contains specifications for dispatch and dynamic semantics (`SCROLLDispatch` in Sect. 9.3.1 and `SCROLLDynamic` in Sect. 9.3.2). This approach hides the complex semantics of role dispatch under only two interfaces rather than scattering them across many interfaces. This layer is static and should only be changed in case one wants to change the semantics of the dynamic dispatch used within the *SCROLL* MOP. The actual configuration of the dispatch is possible, also at runtime as explained in Sect. 9.2.3.

9.3.1. SCROLLDISPATCH

Fully qualified class name
`scroll.internal.Compartment#SCROLLDispatch` (as inner class)

Full source code
See Listing D.6 on page 189

API

```
def dispatch[E](on: Any, m: Method): Either[InvocationError, E]
def dispatch[E, A](on: Any, m: Method, args: Seq[A]):
    Either[InvocationError, E]
```

Table 9.11.: Overview for `SCROLLDispatch`.

This trait implements the actual dispatching for roles as evolving objects, i.e., its methods and functions. The implicit class `Player` (Sect. D.6) uses this implementation directly by extending the trait. An overview of its API can be found in Table 9.11. The following provides a more detailed description:

`def dispatch[E](on: Any, m: Method): Either[InvocationError, E]:`
: This function implements dispatch on an empty argument list for the provided instance `on` (e.g., the player or one of its role instances) and the given method `m`, previously selected by the actual dispatch description `DispatchQuery` (see Sect. 9.2.3) that was specified by the developer. It returns the result of the reflective invocation of the method `m` or an appropriate error (e.g., if `m` was not invokable through the reflection API).

`def dispatch[E, A](on: Any, m: Method, args: Seq[A]): Either[InvocationError, E]:`
: This function implements dispatch on a multi-argument list for the provided instance `on` (e.g., the player or one of its role instances), the given method `m`, previously selected by the actual dispatch description `DispatchQuery` (see Sect. 9.2.3) that was specified by the developer, and its arguments `args`. It returns the result of the reflective invocation of the method `m` or an appropriate error (e.g., if `m` was not invokable through the reflection API).

9.3.2. SCROLLDYNAMIC

Fully qualified class name
`scroll.internal.Compartment#SCROLLDynamic` (as inner class)

Full source code
See Listing D.6 on page 189

API

```
def applyDynamic[E, A](name: String)(args: A*)(implicit dispatchQuery:
    DispatchQuery): Either[SCROLLError, E]
def applyDynamicNamed[E](name: String)(args: (String, Any)*)(implicit
    dispatchQuery: DispatchQuery): Either[SCROLLError, E]
def selectDynamic[E](name: String)(implicit dispatchQuery: DispatchQuery):
    Either[SCROLLError, E]
def updateDynamic(name: String)(value: Any)(implicit dispatchQuery:
    DispatchQuery): Unit
```

Table 9.12.: Overview for SCROLLDynamic.

This section presents the main entry point into *SCROLL*, the `Dynamic` trait that is started by a compiler rewrite of a call to *SCROLL*. This trait enables the dynamic invocation of methods that are not natively available on the player object, but should be found at its roles (see Sect. 8.2.1). An overview of its API can be found in Table 9.12. The following provides a more detailed description:

`def applyDynamic[E, A](name: String)(args: A*)(implicit dispatchQuery: DispatchQuery): Either[SCROLLError, E]`:
This function is called through the compiler rewrite and allows to invoke a function with the given `name` and arguments `args`. By providing an implicit `DispatchQuery` (see Sect. 9.2.3), the user can steer the search for the appropriate receiver of the call (as explained in Sect. 9.3.3). The result of the reflective invocation of the function with the given `name` or an appropriate error (e.g., the function with the given `name` was not invokable through the reflection API) is returned.

`def applyDynamicNamed[E](name: String)(args: (String, Any)*) (implicit dispatchQuery: DispatchQuery): Either[SCROLLError, E]`:
This function is called through the compiler rewrite and allows to invoke a function with the given `name` and named arguments `args`. By providing an implicit `DispatchQuery` (see Sect. 9.2.3), the user can steer the search for the appropriate receiver of the call (as explained in Sect. 9.3.3). The result of the reflective invocation of the function with the given `name` or an appropriate error (e.g., the function with the given `name` was not invokable through the reflection API) is returned.

`def selectDynamic[E](name: String) (implicit dispatchQuery: DispatchQuery): Either[SCROLLError, E]`:
This function is called through the compiler rewrite and allows to get the runtime value of an attribute with the given `name`. By providing an implicit `DispatchQuery` (see Sect. 9.2.3), the user can steer the search for the appropriate receiver of the call (as explained in Sect. 9.3.3). The result of the reflective invocation of the attribute with the given `name` or an appropriate error (e.g., the attribute with the given `name` was not invokable through the reflection API) is returned.

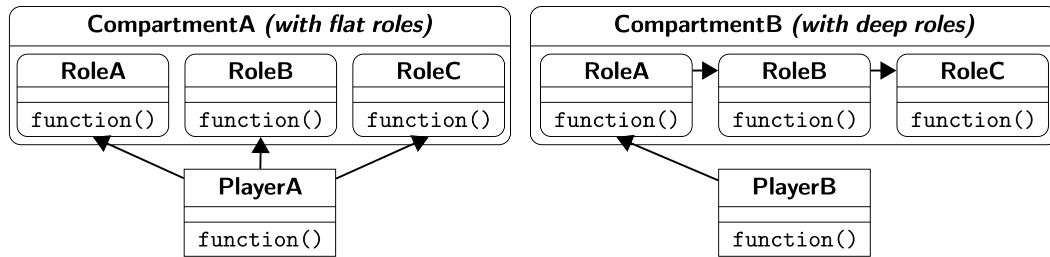


Figure 9.2.: An example for the need of customizable role dispatch. It is ambiguous which role is responsible for answering a call to `function()`. Both flat roles (roles cannot play roles themselves, *left side*) as well as deep roles (*right side*) reveal the same ambiguity here.

```
def updateDynamic(name: String)(value: Any)
    (implicit dispatchQuery: DispatchQuery): Unit:
```

This function is called through the compiler rewrite and allows to set the runtime value of an attribute with the given `name`. By providing an implicit `DispatchQuery` (see Sect. 9.2.3), the user can steer the search for the appropriate receiver of the call (as explained in Sect. 9.3.3).

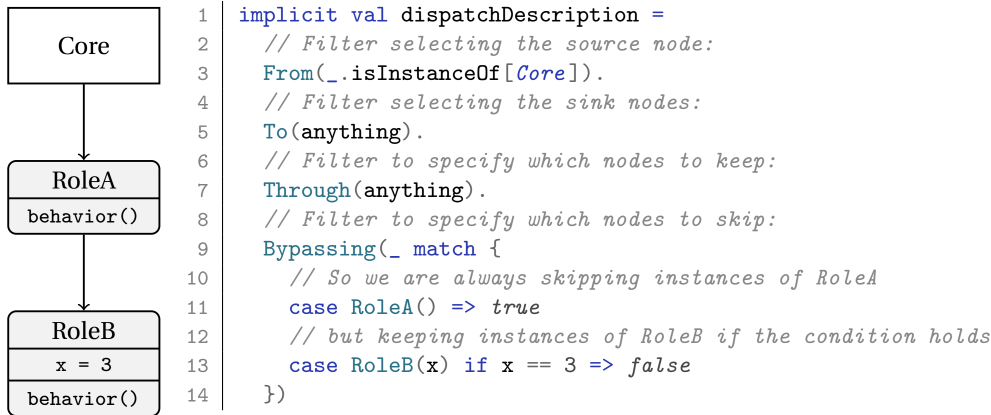
9.3.3. DISPATCH NOTATIONS

Sometimes it is ambiguous or context-dependent which role should be selected for answering a call to the required behavior (see Fig. 9.2). The developer should be able to control the desired selection. *SCROLL* supports this with function composition and Scala's pattern matching making use of an explicit dispatch description which is passed to the actual method invocation as implicit argument. The given filter functions are applied while traversing the role-play graph.

Fig. 9.3 provides an additional example. A new dispatch description is constructed and the appropriate selection functions are passed to the factory methods of the dispatch description. The search starts from the player. Hence, the type of the player (`_.isInstanceOf[Core]`) is used to parameterize the `From`-selector. Other types of roles that are in the role-play graph are of no particular interest. For this, `anything` is passed to the `To`-selector, which will always evaluate to `true`. Then, every role instance will be considered, while traversing the role-play graph. The same parameterization is applied to the `Through`-selector on intermediate nodes. Finally, for the `Bypassing`-selector, only instances of `RoleB` with the state `x = 3` should be selected, so that these should never be bypassed. In summary, with an explicit dispatch description the developer defines a sub-role-graph of the role-graph.

9.3.4. DISPATCH MAPPING TO SCALA

The dispatch mechanism provided by the *SCROLL* MOP is declarative and parameterizable. It allows for four-dimensional, context-aware dispatch as originally introduced for context-oriented programming by Hirschfeld et al. [2008] (see Sect. 4). Instead of only associating the behavior called with a name (first dimension), the receiver context (second dimension), the sender context (third dimension), and the overall system context (fourth dimension) are taken into account. The filter expressions are inspired from adaptive programming, where programs can be understood as normal object-oriented programs, but the class graph is a parameter. Hence, the class graph can be changed during runtime, without changing the actual program [Lämmel et al., 2003]. Adaptive




```
implicit val dispatchDescription =
  // Filter selecting the source node:
  From(_.isInstanceOf[Core]).
  // Filter selecting the sink nodes:
  To(anything).
  // Filter to specify which nodes to keep:
  Through(anything).
  // Filter to specify which nodes to skip:
  Bypassing(_ match {
    // So we are always skipping instances of RoleA
    case RoleA() => true
    // but keeping instances of RoleB if the condition holds
    case RoleB(x) if x == 3 => false
  })
```

Listing 9.3.: Example code for an explicit dispatch description.

Figure 9.3.: Another example for the need of customizable role dispatch.

Notation	**Meaning**
Dispatch Queries	
Θ	Set of selection functions
$\alpha : N \to \{\top, \bot\}$	Selection function with $\alpha \in \Theta$ assigning boolean values to a given node N according to its evaluation during runtime
$\rhd : \hat{\mathcal{P}}(N) \times \Theta \to \hat{\mathcal{P}}(N)$	Dispatch filter selecting the source node of the dispatch problem from set $\hat{\mathcal{P}}(N)$, with regard to the evaluation of the selection function $\alpha \in \Theta$
$\lhd : \hat{\mathcal{P}}(N) \times \Theta \to \hat{\mathcal{P}}(N)$	Dispatch filter selecting sink nodes (given from set $\hat{\mathcal{P}}(N)$) of the dispatch problem, i.e., the potential receivers of messages, with regard to the evaluation of the selection function $\alpha \in \Theta$
$\unrhd : \hat{\mathcal{P}}(N) \times \Theta \to \hat{\mathcal{P}}(N)$	Dispatch filter to specify which nodes to keep in the given set $\hat{\mathcal{P}}(N)$, with regard to the evaluation of the selection function $\alpha \in \Theta$
$\ntrianglerighteq : \hat{\mathcal{P}}(N) \times \Theta \to \hat{\mathcal{P}}(N)$	Dispatch filter to specify which nodes to remove from the given set $\hat{\mathcal{P}}(N)$, with regard to the evaluation of the selection function $\alpha \in \Theta$
$\Omega : \hat{\mathcal{P}}(N) \to \hat{\mathcal{P}}(N)$	Composed dispatch query $\Omega(\hat{\mathcal{P}}(N)) = (\ntrianglerighteq \circ \unrhd \circ \lhd \circ \rhd)(\hat{\mathcal{P}}(N))$, i.e., applying the composition of all dispatch filters above to the set of nodes $\hat{\mathcal{P}}(N)$

Table 9.13.: Notation overview for dispatch queries.

programming utilizes parameterizable traversal specifications. This is directly transferred to the dispatch concept of *SCROLL*: `from` (to identify source nodes), `to` (identifies target nodes), `through` (identifies required intermediate nodes), and `bypassing` (identifies intermediates to be skipped). A detailed overview is given in Table 9.13. The end-user can now freely plug in its specific selection functions with the provided `DispatchQuery` API (see Sect. 9.2.3).

The implementation in Scala is now simple (see Table 9.14). Classes for the aforementioned selector functions ($\rhd$ as `from`, $\lhd$ as `to`, $\unrhd$ as `through`, and $\ntrianglerighteq$ as `bypassing`) are directly mapped to classes inheriting the generic `Function` interface from Scala, incorporating the functional mapping:

$\alpha : N \rightarrow \{\top, \bot\}$:

```scala
f: Any => Boolean
```

$\triangleright : \hat{\mathcal{P}}(N) \times \Theta \rightarrow \hat{\mathcal{P}}(N)$:

```scala
class From(val sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
  override def apply(edges: Seq[Any]): Seq[Any] =
    edges.slice(edges.indexWhere(sel), edges.size)
}
```

$\triangleleft : \hat{\mathcal{P}}(N) \times \Theta \rightarrow \hat{\mathcal{P}}(N)$:

```scala
class To(val sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
  override def apply(edges: Seq[Any]): Seq[Any] =
    edges.lastIndexWhere(sel) match {
    case -1 => edges
    case _ => edges.slice(0, edges.lastIndexWhere(sel) + 1)
  }
}
```

$\trianglerighteq : \hat{\mathcal{P}}(N) \times \Theta \rightarrow \hat{\mathcal{P}}(N)$:

```scala
class Through(sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
  override def apply(edges: Seq[Any]): Seq[Any] = edges.filter(sel)
}
```

$\ntrianglerighteq : \hat{\mathcal{P}}(N) \times \Theta \rightarrow \hat{\mathcal{P}}(N)$:

```scala
class Bypassing(sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
  override def apply(edges: Seq[Any]): Seq[Any] = edges.filterNot(sel)
}
```

$\Omega : \hat{\mathcal{P}}(N) \rightarrow \hat{\mathcal{P}}(N)$:

```scala
class DispatchQuery(from: From, to: To, through: Through, bypassing:
  Bypassing) {
  def filter(anys: Seq[Any]): Seq[Any] =
    from.andThen(to).andThen(through).andThen(bypassing)(anys.distinct)
}
```

Table 9.14.: Mapping from dispatch functions to the Scala implementation.

$$\hat{\mathcal{P}}(N) \rightarrow \hat{\mathcal{P}}(N) \rightsquigarrow \texttt{Seq[Any] => Seq[Any]}$$

The user-defined selection function $\alpha \in \Theta$ is applied as:

$$\alpha \rightsquigarrow \texttt{Any => Boolean}$$

At the end, the final query Ω is composed with:

$$\Omega(\hat{\mathcal{P}}(N)) = (\ntrianglerighteq \circ \trianglerighteq \circ \triangleleft \circ \triangleright)(\hat{\mathcal{P}}(N))$$

Here, Scala's function composition with andThen is used:

$$\Omega(\hat{\mathcal{P}}(N)) \rightsquigarrow \texttt{from.andThen(to).andThen(through).andThen(bypassing)}(E)$$

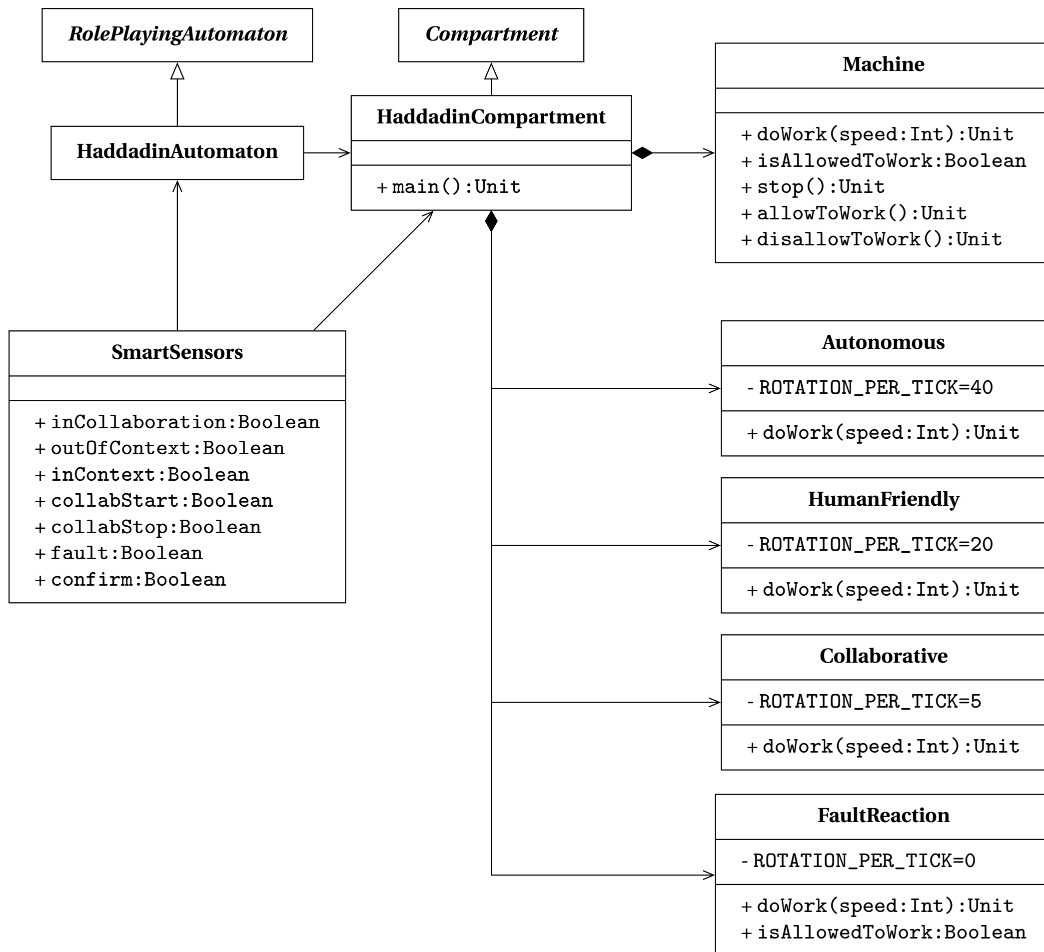


Figure 9.4.: Class diagram for the robotic co-worker example.

9.4. PROGRAMMING THE ROBOTIC CO-WORKER WITH *SCROLL*

This section shows how the robotic co-worker example introduced in Sect. 6 may be implemented within *SCROLL*. As a reference, inspect the class diagram presented in Fig. 9.4. Additionally, the full code can be found as follows:

Machine Listing D.1 on page 183

HaddadinAutomaton Listing D.2 on page 184

HaddadinCompartment Listing D.3 on page 186

SmartSensors Listing D.4 on page 187

9.4.1. THE CLASS MACHINE

The class `Machine` is implementing the core behavior with a simple workflow chaining together picking up (`pickBox`), handling (`handleBox`), and placing a box (`placeBox`) with a predefined speed (Listing 9.4). This speed value will be the target for adaptation when switching between different modes of autonomy with regard to the human presence (Fig. 6.2).

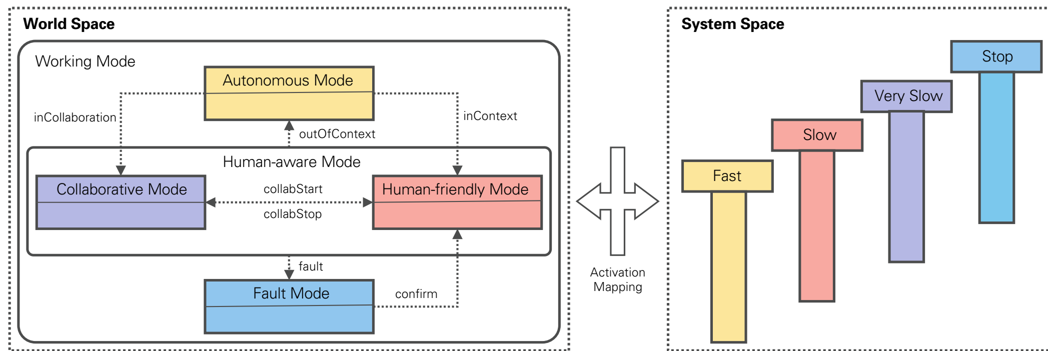


Figure 6.2.: The mapping from the Haddadin world space to the system space (from page 48).

```
case class Machine() {
  private var allowedToWork = true

  private def pickBox(speed: Int): Unit = { /* ... */ }

  private def handleBox(speed: Int): Unit = { /* ... */ }

  private def placeBox(speed: Int): Unit = { /* ... */ }

  // simple, simulated workflow
  def doWork(speed: Int): Unit = {
    // picking up a box with a certain speed
    pickBox(speed)
    // handling or inspecting this box with a certain speed
    handleBox(speed)
    // and placing it somewhere with a certain speed
    placeBox(speed)
  }

  def isAllowedToWork: Boolean =
    allowedToWork

  def stop(): Unit =
    disallowToWork()

  def allowToWork(): Unit = {
    this.allowedToWork = true
  }

  def disallowToWork(): Unit = {
    this.allowedToWork = false
  }
}
```

Listing 9.4.: Source code excerpt for `Machine`.

9.4.2. THE CLASS HADDADINAUTOMATON

The `HaddadinAutomaton` (Listing 9.5) realizes the automaton in Fig. 6.2. It handles the actual adaptation by adding and removing the appropriate roles to the `Machine`. All required events and states are predefined in its companion object (see Appendix B). The reaction to certain incoming events (e.g., `fault`, or `in_perception`) is caught by the `when` clauses (Line 23). With the definition of the `onTransition` clause (Line 36), the actual role binding and unbinding takes place with regard to the specific state transitions.

```
object HaddadinAutomaton {
  // Events:
  case object E_collaborate extends RPAData
  case object E_confirm extends RPAData
  /* ... */

  // States:
  case object Autonomous_Mode extends RPAState
  case object Collaborative_mode extends RPAState
  /* ... */
}

class HaddadinAutomaton(val comp: HaddadinCompartment) extends
    RolePlayingAutomaton {
  // Core:
  val machine = comp.machine

  // Roles:
  val machine_autonomous = comp.machine_autonomous
  val machine_collaborative = comp.machine_collaborative
  val machine_faultreaction = comp.machine_faultreaction
  val machine_humanfriendly = comp.machine_humanfriendly

  when(Autonomous_Mode) {
    case Event(E_fault, _) => goto(Fault_reaction_autonomous)
    case Event(E_in_perception, _) => goto(Human_friendly_mode)
  }

  when(Collaborative_mode) {
    case Event(E_fault, _) => goto(Fault_reaction_collaborative)
    case Event(E_out_perception, _) => goto(Autonomous_Mode)
    case Event(E_stop_collaborate, _) => goto(Human_friendly_mode)
  }

  /* ... */

  onTransition {
    case Start -> Autonomous_Mode =>
      comp.addPlaysRelation(machine, machine_autonomous)
    case Autonomous_Mode -> Fault_reaction_autonomous =>
      comp.removePlaysRelation(machine, machine_autonomous)
      machine.stop()
      comp.addPlaysRelation(machine, machine_faultreaction)
    case Autonomous_Mode -> Human_friendly_mode =>
      comp.removePlaysRelation(machine, machine_autonomous)
      comp.addPlaysRelation(machine, machine_humanfriendly)
    /* ... */
  }
}
```

Listing 9.5.: Source code excerpt for `HaddadinAutomaton`.

9.4.3. THE CLASS HADDADINCOMPARTMENT

The class `HaddadinCompartment` (Listing 9.6) contains the implementation of the roles for adapting the overall robot behavior represented by the player class `Machine`. The roles `Autonomous` (Line 12), `HumanFriendly` (Line 23), `Collaborative` (Line 33), and `FaultReaction` (Line 43) are simply passing the call for invoking the workflow to its core (i.e., a `Machine` instance) while adapting the systems speed (`ROTATION_PER_TICK`).

```
1  class HaddadinCompartment extends Compartment {
2    // Player:
3    val machine = Machine()
4
5    // Roles:
6    val machine_autonomous = Autonomous()
7    val machine_collaborative = Collaborative()
8    val machine_faultreaction = FaultReaction()
9    val machine_humanfriendly = HumanFriendly()
10   /* ... */
11
12   case class Autonomous() {
13
14     private val ROTATION_PER_TICK = 40
15
16     def doWork(speed: Int): Unit = {
17       // bypassing the role itself so doWork is not called recursively
18       implicit val dd = Bypassing(_.isInstanceOf[Autonomous])
19       (+this).doWork(ROTATION_PER_TICK)
20     }
21   }
22
23   case class HumanFriendly() {
24
25     private val ROTATION_PER_TICK = 20
26
27     def doWork(speed: Int): Unit = {
28       implicit val dd = Bypassing(_.isInstanceOf[HumanFriendly])
29       (+this).doWork(ROTATION_PER_TICK)
30     }
31   }
32
33   case class Collaborative() {
34
35     private val ROTATION_PER_TICK = 5
36
37     def doWork(speed: Int): Unit = {
38       implicit val dd = Bypassing(_.isInstanceOf[Collaborative])
39       (+this).doWork(ROTATION_PER_TICK)
40     }
41   }
42
43   case class FaultReaction() {
44
45     private val ROTATION_PER_TICK = 0
46
47     def doWork(speed: Int): Unit = { // we do nothing }
48     def isAllowedToWork: Boolean = false
49   }
50 }
```

Listing 9.6.: Source code excerpt for `HaddadinCompartment`.

9.4.4. THE CLASS SMARTSENSORS

The class `SmartSensors` (Listing 9.7) pushes events to the `HaddadinAutomaton` generated every 100 millisecond with regard to the human presence. Finally, this leads to the role-based adaptation of the overall system. The functional modes are represented and implemented as roles overriding their players behavior.

```
1  class SmartSensors(comp: HaddadinCompartment) {
2    val automaton = new HaddadinAutomaton(comp)
3
4    def workerIsIn(room: Room): Boolean = worker.isIn(room)
5    def inCollaboration: Boolean = workerIsIn(machineArea)
6    def outOfContext: Boolean = workerIsIn(roomA)  workerIsIn(exit)
7    def inContext: Boolean = !outOfContext
8    def collabStart: Boolean = inCollaboration
9    def collabStop: Boolean = !inCollaboration
10   def fault: Boolean = workerIsIn(machine)
11   def confirm: Boolean = !fault
12
13   val constraintsActor = new Actor {
14     def receive = {
15       case Check =>
16         if (fault) {
17           // the method ! is used to send events to the object representing
               the role-playing automaton
18           automaton ! E_fault
19         }
20         if (confirm) {
21           comp.machine.allowToWork()
22           automaton ! E_confirm
23         }
24         if (outOfContext) {
25           automaton ! E_out_perception
26         }
27         if (inContext) {
28           automaton ! E_in_perception
29         }
30         if (collabStart) {
31           automaton ! E_collaborate
32         }
33         if (collabStop) {
34           automaton ! E_stop_collaborate
35         }
36     }
37   }
38
39   def start(): Unit = {
40     schedule(0 seconds, 100 milliseconds, constraintsActor, Check)
41     automaton ! Uninitialized
42   }
43 }
```

Listing 9.7.: Source code excerpt for `SmartSensors`.

The actual adaptation logic (*when* to bind and unbind those roles) and the adaptation itself (*what* and *how* to adapt) are clearly separated, leading to an increased separation of concerns, and, ultimately, better maintainability and extensibility in the case of unforeseen future needs for adaptation. These advantages are enabled by specifying context-dependent behavior and structure in separate role types. Structured contexts group those role types to specific adaptations. Furthermore, the syntax of Scala is untouched.

10

TECHNICAL LIMITATIONS

10.1. LIMITATIONS AND ALTERNATIVES

SCROLL allows for role-based programming with the concept of dynamically evolving objects and purely embeds roles in a statically typed, object-oriented host language. This supports the developer with the best of both worlds: static typing leads to an earlier detection of programming mistakes through static code analysis, better documentation in form of type-signatures, compiler-optimization, runtime-efficiency and an improved design-time development experience, while dynamic objects support easy prototyping, change to unknown requirements or unpredictable data and application integration. Nevertheless, implemented as an library approach on-top of the Scala programming language, there exists no built-in abstraction of those dynamically evolving objects on type level yet. Hence, the following major limitations apply.

10.1.1. LIMITED TYPE-SAFETY

SCROLL uses Scala's `Dynamic` trait [EPFL, 2016b] to address all dynamic behavior from roles that is not available at the player. This is comparable to the usual implementations of dispatch tables (e.g., with C++ vtable, or Java call-sites). Calls to role-specific functionality that would normally fail during type checking phase of Scala are rewritten *after* the typing phase of the Scala compiler (see Table 10.1). At this point, type-safety is lost. The actual set of roles as dynamic extensions that are bound to the player is not statically known, hence static type-safety is not available. At runtime, compound objects representing role-playing, dynamic objects are always represented as `Player[T]` (as explained in Sect. 9.2.2), where `T` refers to the type of the player. No special typing construct is available to mirror role-playing objects in first place, hence calls via the `Dynamic` trait cannot be statically typed. To remedy this shortcoming, and to help the developer, providing additional warnings and error messages whenever the requested dynamic behavior is unlikely to exist at all, the *SCROLLCompilerPlugin* was developed (see Sect. 10.2). The following approaches for generating actual role-specific types in the context of role-playing objects may be considered for future work:

InvokeDynamic `Invokedynamic` is a bytecode instruction for the JVM that facilitates the implementation of dynamic languages through dynamic method invocation. In a dynamic language, type checking typically occurs at runtime. Developers must pass appropriate types or risk runtime failures. Bytecode can be generated at runtime to weave-in the `invokedynamic` bytecode instruction replacing any of the four method-invocation instructions: *invokestatic* is used to invoke static methods, *invokevirtual* to invoke public and protected non-static methods, *invokeinterface* with method dispatch being based on a interface type, and finally, *invokespecial* to invoke instance initialization methods as well as private and superclass methods. To address poor performance, the `invokedynamic` instruction eliminates the need for ad-hoc runtime support. Instead, the call bootstraps by invoking runtime logic that efficiently selects a target method, and subsequent calls typically invoke the cached target methods without having to re-bootstrap. Dynamic languages profit from `invokedynamic` because it supports dynamically changing call site targets.

A call site, more specifically, a dynamic call site is an `invokedynamic` instruction. Furthermore, because the JVM internally supports `invokedynamic`, this instruction can be better optimized by the Just-In-Time (JIT) compiler. A *method handle*

method handle

ID	Phase name	Description
1	parser	parse source into ASTs, perform simple desugaring
2	namer	resolve names, attach symbols to named trees
3	packageobjects	load package objects
4	**typer**	**the meat and potatoes: type the trees**
5	patmat	translate match expressions
6	superaccessors	add super accessors in traits and nested classes
7	extmethods	add extension methods for inline classes
8	pickler	serialize symbol tables
9	refchecks	reference/override checking, translate nested objects
10	uncurry	uncurry, translate function values to anonymous classes
11	fields	synthesize accessors and fields, add bitmaps for lazy vals
12	tailcalls	replace tail calls by jumps
13	specialize	@specialized-driven class and method specialization
14	explicitouter	this refs to outer pointers
15	erasure	erase types, add interfaces for traits
16	posterasure	clean up erased inline classes
17	lambdalift	move nested functions to top level
18	constructors	move field definitions into constructors
19	flatten	eliminate inner classes
20	mixin	mixin composition
21	cleanup	platform-specific cleanups, generate reflective calls
22	delambdafy	remove lambdas
23	jvm	generate JVM bytecode
24	terminal	the last phase during a compilation run

Table 10.1.: The Scala compiler phases (available with `scalac -Xshow-phases`).

is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. It is similar to a C-style function pointer that points to executable code and which can be dereferenced to invoke this code. At runtime, an `invokedynamic` call site is bound to a method handle by a bootstrap method. This method is executed the first time the JVM encounters this call site during execution. The `invokedynamic` instruction is followed by an operand that serves as an index into the constant pool of a classfile. The call site instance is said to contain the method handle and becomes permanently linked to the call site (the `invokedynamic` instruction). Although the call site instance refers to the same program location throughout its lifetime, it may allow its target method handle to be redefined during code execution. Is sum, that concept provides a solid base for implementing the role-specific adding, removal or transfer of context-specific behavior and structure and fits perfectly to the *SCROLL* approach.

Scala Macros At first glance, macro definitions in Scala are equivalent to normal function definitions, except for their body, which starts with the conditional keyword `macro` and is followed by a possibly qualified identifier that refers to a static macro implementation method. If, during type checking, the compiler encounters an application of the macro, it will expand that application by invoking the corresponding macro implementation method, with the abstract-syntax trees of the argument expressions as arguments. The result of the macro implementation is

another abstract syntax tree, which will be inlined at the call site and will be type-checked in turn. With that, all API calls within *SCROLL* could be replaced by macro definitions rewriting the current tree to a role-aware typing construct. This goes beyond the error message reporting from the ScalaCompilerPlugin presented in Sect. 10.2.

Scala Compiler Plugin A compiler plugin allows to modify the behavior of the compiler itself without needing to change the main Scala distribution. Usually, this does not happen very frequently, because Scala's light, flexible syntax will allow for a better solution using, e.g, a library. Nevertheless, there are use-cases where a compiler modification is the best choice even for Scala: adding additional compile-time checks, adding compile-time optimizations for a heavily used library, or rewriting the Scala syntax into an entirely different, custom meaning. Adding a phase for extra checks and extra tree rewrites that apply after type checking, would allow to generate a role-aware typing construct and would fit perfectly to the *SCROLL* approach.

Custom Type System Finally, one could implement a completely new type system with one of the meta-programming systems, e.g., with *Intellij MPS* [JetBrains, 2017], *Xsemantics* for *Xtext* [Efftinge and Völter, 2006], or *TS* for *Spoofax* [Kalleberg et al., 2007]. With that, further research on such a type system needs to happen, e.g., with regard to type inferencing rules, reduction rules, and its soundness.

10.1.2. PERFORMANCE ISSUES

As the quantitative evaluation (see Sect. 11.2) shows, *SCROLL* performs quite slowly due to the heavy use of the Java Reflection API. Because reflection involves types that are dynamically resolved, most of the Java virtual machine optimizations cannot be applied. Many additional tasks need to be performed from the JVM while using reflection: checking that there is a parameterless constructor, checking the accessibility of that parameterless constructor, checking that the caller has access to use reflection at all, calculate how much space needs to be allocated during runtime, and make calls into the constructor code because it is unknown beforehand if the constructor is empty. Consequently, reflective operations are slower than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications. In the scope of this thesis that is not to be considered critical, as it serves as a testbed for dynamic dispatch. Nevertheless, with the approaches mentioned in the section above (Sec 10.1.1) for implementing suitable, role-aware types (`invokedynamic` on the JVM, Scala macros, or with a Scala compiler plugin) the overall performance could be improved tremendously. This is considered to be out of the scope for this thesis, but targeted as future work (see Sect. 16).

10.2. THE *SCROLL* COMPILER PLUGIN

As soon as the compiler triggers its rewrite rules (Scala's `Dynamic` as explained in Sect. 8.2.1) certain type-safety is lost (see Listing 10.1 Line 17 and 18) because it cannot be statically determined if a role is actually bound during runtime. FRaMED [Kühn et al., 2016] is able to export instances of CROM [Kühn et al., 2014] as Ecore files serialized as XMI. With the help of such an optionally imported file the *SCROLLCompilerPlugin* [Leuthäuser, 2016] will check all of the statements invoking role calls (e.g., `applyDynamic` or `selectDynamic`) against the available player and role classes from

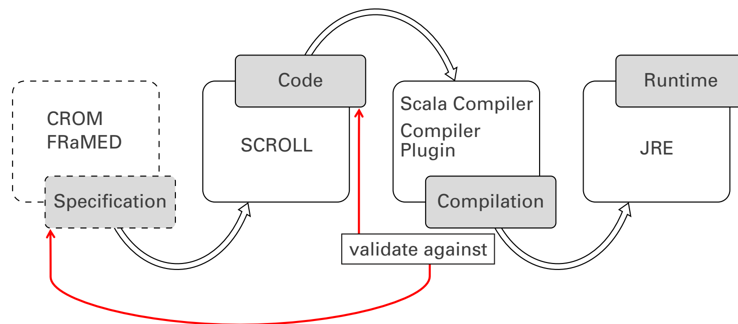


Figure 10.1.: The CompilerPlugin toolchain.

the model instance and the class definitions in the current scope. In the example, the behavior of the compound robot is aggregated at runtime (Listing 10.1, Line 15) from multiple roles (`Navigation` for finding a target, `Observer` for reading sensor values and `Vehicle` for the actual movement). Because it cannot be statically determined if the corresponding role is actually bound, type-safety is lost when the Scala compiler applies its rewrite rules. The SCROLLCompilerPlugin generates meaningful messages and reports them as warnings or compile time errors (which is configurable) to the developer (see Listing 10.2).

```
1  case class Robot(name: String)
2  case class Service() {
3    def move() {
4      val name: String = +this name()
5      val target: String = +this getTarget()
6      val sensorValue: Int = +this readSensor()
7      val actor: String = +this getActor()
8      /* do something with that values */
9    }
10 }
11 case class Navigation() { def getTarget = "kitchen" }
12 case class Observer() { def readSensor = 100 }
13 case class Vehicle() { def getActor = "wheels" }
14
15 val kuka = Robot("Kuka") play Service() play Navigation() play Observer() play
     Vehicle()
16
17 kuka move()
18 kuka swim()
```

Listing 10.1.: Example for the application of the SCROLLCompilerPlugin.

```
1 [error] Example.scala:29: applyDynamic as 'swim' detected on: 'Robot'.
2 [error]   For 'Robot' the following dynamic extensions are specified:
3 [error]     - 'Robot' -> 'Service', 'Navigation', 'Observer', 'Vehicle'
4 [error]           kuka swim()
5 [error]                ^
6 [error] Example.scala:29: Neither 'Robot', nor its dynamic extensions offer
    the called behavior! This may indicate a programming error!
7 [error]           kuka swim()
8 [error]                ^
```

Listing 10.2.: Console output of the SCROLLCompilerPlugin for the robot example.

To gather the behavior offered in all possibly attached roles as dynamic extensions, all relevant binding- and unbinding statements, player classes and their behavior, and all calls to the `Dynamic` trait, the Scala Abstract Syntax Tree (AST) is traversed at compile-time right after the typer phase of the standard Scala compiler [Lightbend Inc., 2016b].

Algorithm 1 Collection of all relevant binding- and unbinding statements, player classes and their behavior, and all calls to the `Dynamic` trait in *SCROLL* programs within the *SCROLLCompilerPlugin*.

```
1:  DS ← {ApplyDynamic, ApplyDynamicNamed, SelectDynamic, UpdateDynamic}
2:  BS_Play, BS_Transfer, BS_Remove, P, PM, B, BM, D ← ∅          ▷ Initialization
3:  procedure COLLECT(tree: ScalaAST)
4:      if tree = ps and ps is a play statement then
5:          BS_Play ← BS_Play ∪ {ps}
6:      end if
7:      if tree = ts and ts is a transfer statement then
8:          BS_Transfer ← BS_Transfer ∪ {ts}
9:      end if
10:     if tree = ds and ds is remove statement then
11:         BS_Remove ← BS_Remove ∪ {ds}
12:     end if
13:     if tree = pc and pc is a player class then
14:         P ← P ∪ {pc} and PM ← PM ∪ {pc.name → pc}
15:     end if
16:     if tree = b and b is some behavior then
17:         B ← B ∪ {b} and BM ← BM ∪ {b.signature → b}
18:     end if
19:     if tree = d and d ∈ DS then
20:         D ← D ∪ {d}
21:     end if
22:     if tree was not covered by previous cases then
23:         COLLECT(tree.subtree)          ▷ call COLLECT recursively on the subtree
24:     end if
25: end procedure
```

This algorithm is shown in more detail in Algorithm 1. `COLLECT` is a program analysis which runs on the AST until all AST subtrees are covered. It uses the following sets:

- BS_{Play}: Binding statements for play relationship.
- BS_{Transfer}: Binding statements for the transfer of a dynamic extension between objects.
- BS_{Remove}: Binding statements for the removal of a dynamic extension from an object.
- P: All available player classes.
- PM : Name → P: Mapping from the name of a player to its class.
- B: Available behavior for a player (i.e., its methods / functions, and attributes).
- BM : Signature → B: Mapping from the signature of a behavior to the behavior itself.
- D: Calls to the `Dynamic` trait.

After evaluating `COLLECT` at compile-time, the aforementioned sets of statements and collected behavior are compared against each other and warnings or errors will be reported to the user. An overview of the resulting tool chain is visualized in Fig. 10.1.

11

EVALUATION

11.1. QUALITATIVE EVALUATION

The qualitative evaluation is split into three parts. First, we analyze the fulfillment of the requirements stated in Sect. 7 (Sect. 11.1.1). Secondly, an analysis of *SCROLL* based on a previously defined classification scheme [Kühn et al., 2014] is given where we check the fulfillment of each feature (Sect. 11.1.2). Finally, the variability analysis from Graversen [2006] is adapted and applied to *SCROLL* as a third evaluation (Sect. 11.1.3).

11.1.1. A REQUIREMENT-BASED ANALYSIS FOR ROLES WITH *SCROLL*

This section summarizes how the requirements derived in Sect. 7 are implemented with the *SCROLL* approach. The comparison with contemporary approaches from the literature can be found in Table 11.1.

/F.1/ *No additional tooling*
SCROLL implemented as library in Scala does not require any additional tooling (i.e., a custom compiler or development environment), the standard Scala compiler is sufficient without any modifications. No code is generated, hence all debuggers delivered e.g., with the ScalaIDE (based on Eclipse) or IntelliJ remain useful. All the other surveyed role-based programming languages are either extensions to the Java language, requiring an adapted compiler, or use a generative approach. ScalaRoles is the only remarkable exception here as it is implemented as a library as well. Thus, it can be seen as the predecessor in spirit of *SCROLL*. The concept of the compound object stems from ScalaRoles, although it is technically different (dynamic proxies versus `Dynamic` trait with compiler rewrites and implicits).

/F.2/ *Dispatch configurable at runtime*
With dispatch descriptions as first-class entities configurable by the user itself (defining a `DispatchQuery`, see Sect. 9.2.3), *SCROLL* is the only available implementation allowing for fully customizable dispatch at runtime. In OT/J, one could exploit its explicit team activation and deactivation concept guarded by writing constraints, although this requires additional management code and leads to a lower maintainability. The same holds for re-configuring dynamic proxy creation in ScalaRoles.

/F.3.1/ *Multi-dimensional dispatch: Associate the computational unit with a name*
As all implementations investigated here are either extensions to the Java language, or generate Java code directly, a computational unit (i.e., Java method and attribute) is associated with a name.

/F.3.2/ *Multi-dimensional dispatch: Take the receiver context into account*
Furthermore, the receiver, i.e., its type is taken into account during the dispatch.

/F.3.3/ *Multi-dimensional dispatch: Take the sender context into account*
With a `DispatchQuery`, *SCROLL* takes the sender context into account as well. With the lifting mechanism provided by OT/J, this is also possible as lifting requires three pieces of information: a base instance, a team instance and a required role type.

/F.3.4/ *Multi-dimensional dispatch: Take the system context into account*
Finally, the same holds for handling the system context during dispatching.

/F.4/ *Increase modularity through role-based programming*
All investigated competing approaches provide means to handle role-based programming. Nevertheless, non of them incorporates all role-related features as surveyed in Sect. 13.2.2.

/S.1/ *Declarative and parameterizable dispatch description*
Since none of the contemporary approaches offers a configurable dispatch at runtime, no other language besides from *SCROLL* implements a declarative and parameterizable dispatch description.

/S.2/ *Easy to use programming model and API*
Scala, with its highly flexible and adaptable syntax, can be considered as natural testbed for developing domain specific languages, hence the role-related API within *SCROLL* or ScalaRoles are usable fairly straight forward even for inexperienced developers.

/S.3/ *Reasonable performance / scalability*
As argued in Sect. 11.2, with the heavy use of the Java reflection API, *SCROLL* does not provide the scalability needed for performance-sensitive applications. Even the OT/J approach with dynamic bytecode weaving or ScalaRoles dynamic proxy generation can be considered as being too slow.

/S.4/ *Integration in existing tool-chains*
As most of the surveyed role-based programming languages are extensions to the Java language, require an adapted compiler, or use a generative approach, the integration in existing tool-chains is difficult. ScalaRoles, as being a library approach like *SCROLL*, is the only notable exception here. Furthermore, the Eclipse plugin available for OT/J offers a step towards a clean integration into development environment Eclipse.

/S.5/ *Integration / compatibility with existing legacy code*
As most of the surveyed role-based programming languages use a generative approach, the integration of legacy code requires writing adapters or proxies and management code. This is technically no problem at all, but implies some additional burden to the developer. OT/J allows for the adaptation of an already existing code base with byte code weaving in the style of aspect-oriented programming. *SCROLL* and ScalaRoles allow for a seamless integration as being standard Scala code running on the JVM.

/S.6/, /S.7/ *High maintainability and extensibility*
SCROLL and ScalaRoles are the only implementations small enough to be considered as being easily maintainable and extensible. For instance, the mature OT/J tool requires to maintain several thousands lines of code for byte code weaving and the integration of various frameworks, such as the Eclipse environment, with the implementation of its role semantics well hidden into this boilerplate code. Hence, modifications are hard to achieve.

The evaluation with regard to the derived requirements clearly shows the various advantages of *SCROLL*, as it is able to implement all of them, only failing at one (the required performance as explained in Sect. 11.2).

Requirement	**Chameleon** [Østerbye, 2003]	**OT/J** [Herrmann, 2005]	**Rava** [He et al., 2006]	**powerJava** [van der Torre, 2006]	**Rumer** [Balzer et al., 2007]	**ScalaRoles** [Pradel and Odersky, 2008]	**NextEJ** [Kamina and Tamai, 2009]	**JavaStage** [Barbosa and Aguiar, 2012]	**SCROLL**
/F.1/	□	□	□	□	□	■	□	□	■
/F.2/	□	⊞	□	□	□	⊞	□	□	■
/F.3.1/	■	■	■	■	■	■	■	■	■
/F.3.2/	■	■	■	■	■	■	■	■	■
/F.3.3/	□	■	□	□	□	□	□	□	■
/F.3.4/	□	■	□	□	□	□	□	□	■
/F.4/	■	■	■	■	■	■	■	■	■
/S.1/	□	□	□	□	□	□	□	□	■
/S.2/	■	■	■	■	⊞	■	■	■	■
/S.3/	?	⊞	?	?	?	⊞	?	?	□
/S.4/	□	⊞	□	□	□	■	□	□	■
/S.5/	⊞	■	⊞	⊞	□	■	⊞	⊞	■
/S.6/	□	□	□	□	□	■	□	□	■
/S.7/	□	□	□	□	□	■	□	□	■

Table 11.1.: Comparison with contemporary approaches with regard to the requirements from Sect. 7. It differentiates between fully (■), partly (⊞), and not implemented (□) requirements. A question mark (?) denotes that this requirement could not be evaluated, e.g., because no running compiler was available.

11.1.2. A FEATURE-BASED ANALYSIS FOR ROLES WITH *SCROLL*

It is necessary to investigate how *SCROLL* compares to contemporary approaches. Thus, we use the previously defined classification scheme for checking the fulfillment of each of the 26 classifying features of roles from Kühn et al. [2014] (see Table 3.1 on page 28).

1. *Roles have properties and behaviors.*
   Any class can be used as a role in *SCROLL*. It is sufficient to attach variables and methods or functions to a role class.

2. *Roles depend on relationships.*
   There is no mandatory restriction to use relationships between roles or role-playing objects in *SCROLL*. But they can be created at any time within a compartment.

3. *Objects may play different roles simultaneously.*
   By calling the `play`-method multiple times, one can attach as many roles as needed.

4. *Objects may play the same role (type) several times.*
   One may simply instantiate a role multiple times and attach it calling the `play`-method.

5. *Objects may acquire and abandon roles dynamically.*
   One may use the `play`- and `drop`-method to acquire and abandon roles.

6. *The sequence of role acquisition/removal may be restricted.*
   Although, there is no direct support in *SCROLL* for this, one can implement such constraints as plain Scala code between role acquisition and removal. Additionally, *SCROLL* supports defining custom sequences of role acquisition and removal using the `RolePlayingAutomaton` (see Sect. 9.2.10).

7. *Unrelated objects can play the same role.*
   This is possible by using the `play`-method for the same instance of a role class.

8. *Roles can play roles.*
   This is possible by using the `play`-method. It leads to deep roles.

9. *Roles can be transferred between objects.*
   This is possible by using the `transfer`-method.

10. *The state of an object can be role-specific.*
    Role types are implemented with classes or case classes and their state is instance specific. By merging the roles instances with a player object with the `play`-method, they become a compound object. This compound object contains the merged state of the player and its roles.

11. *Features of an object can be role-specific.*
    Different roles (may) have (different) properties and behavior (see role feature 1) and attaching them to players will create a (compound-) object aggregating all those role-specific features.

12. *Roles restrict access.*
    When referencing roles via variables, one only has access to their specific properties and behavior, regardless of any active role-playing relation. On the other hand, using the compound object, one has access to all the features of all roles, regardless if they are reachable from the player or an attached role.

13. *Different roles may share structure and behavior.*
    Role types in *SCROLL* are standard Scala classes or case classes, which can inherit from each other.

14. *An object and its roles share identity.*
    The compound object will always have the same identity as its contained player object and every role object attached.

15. *An object and its roles have different identities.*
    The standard case is the semantics of role feature 14. Nevertheless, one can always override the `equals`- and `hashCode`-functions of the role classes to provide custom identity management.

16. *Relationships between roles can be constrained.*
    Since there are no first-class relationships, no constraints can be applied.

17. *There may be constraints between relationships.*
    Since there are no first-class relationships, no constraints can be applied.

18. *Roles can be grouped and constrained together.*
    They can be grouped into compartments and role groups.

19. *Roles depend on compartments.*
    Technically, one can import roles from anywhere using Scala's `import` keyword. Implementing them directly in compartments might be beneficial for name space management.

20. *Compartments have properties and behaviors.*
    Since compartments can be implemented as traits, classes, or case classes, they can have properties (as class attributes) and behavior (as arbitrary functions or methods).

21. *A role can be part of several compartments.*
    Because one can import roles from anywhere using Scala's `import` keyword, integrating them in multiple, nested compartments is no problem.

22. *Compartments may play roles like objects.*
    Objects of any type are allowed to play roles.

23. *Compartments may play roles which are part of themselves.*
    There are no restrictions concerning the type of the role a compartment might play.

24. *Compartments can contain other Compartments.*
    Hierarchically nested compartments are supported.

25. *Different compartments may share structure and behavior.*
    Since compartments are standard Scala traits, classes or case classes, they support inheritance.

26. *Compartments have their own identity.*
    Since compartments are standard Scala traits, classes or case classes, they carry their own identity.

To investigate how well the implementation with *SCROLL* blends into contemporary approaches, the previously defined scheme from Kühn et al. [2014] with 26 classifying features of roles was investigated. *SCROLL* fully implements 22 of them.

11.1.3. A VARIABILITY ANALYSIS FOR ROLES WITH *SCROLL*

While being fully aware that the role requirements covered by the variability analysis presented in Graversen [2006], as introduced in Appendix A, are both insufficient and imprecise, as stated in by the authors themselves, up to our knowledge, it is the only survey available solely focusing on the features of roles at runtime. In Sect. 3.7, we derived a list of features for roles at runtime (see Table 3.3), extending the lists presented in Steimann [2000b]; Kühn et al. [2014] (see Fig. 3.1). The following shows how *SCROLL* supports the features presented there.

27. *The amount of roles an instance of a class and a role can play may be constrained.*
    Both is supported by *SCROLL*. The first runtime feature requires runtime checking by using `RoleGroups` and their constraints on the contained roles (see Sect. 9.2.9). As a standard within *SCROLL*, instances of a class are not subject to any restrictions on the amount of roles they are allowed to play. The latter runtime feature is supported as well. Both flat roles and deep roles can be implemented. By default, there exists no constraint on the number of roles a player may acquire.

	Feature		Example
27.	The amount of roles an instance of a class and a role can play may be constrained.	■	Sect. 9.2.9
28.	Each role type played must be unique.	■	Sect. 9.2.9
29.	Possible supertypes for classes can be class types, role types, or compound object types.	□	-
30.	The amount of simultaneously existing instances of a role type may be constrained.	⊞	-
31.	The amount of players, a role is played by, may be constrained.	⊞	-
32.	The visibility of roles during dispatching may be constrained.	□	-
33.	Role types are supertypes, subtypes, or unrelated types of their player.	■	Listing 8.6
34.	Role types may extend role types, class types, or compound objects.	■	Listing B
35.	The player type for a role type may be a role type, a class type, an interface type, a metaclass, a compound object type, a property, or undefined.	■	Listing 8.6
36.	Properties of roles can be fields, methods, class methods, and static methods.	■	Listing 9.3
37.	Roles can have nested methods, roles and classes.	■	Listing 9.6
38.	Role instances can be referenced directly, or indirectly.	⊞	Listing 9.5
39.	A reference to a role always points to the compound object.	■	Listing 8.8
40.	Method dispatch on roles happens on the sender or its player, the receiver or its player, the context, or the compound object.	■	Listing 9.3
41.	Self may refer to dual self, or non-virtual self.	■	Listing 8.6
42.	Super refers both to the static inheritance chain, and to the attached roles.	□	-
43.	The player may be referenced directly, or indirectly.	■	Listing 8.6
44.	A role may be called from its player.	■	Listing 8.6
45.	Roles may call among each other.	■	Listing 8.6
46.	Roles may incorporate around-methods.	□	-
47.	Role creation, attachment, and movement may be restricted.	⊞	Listing 9.5
48.	Roles may be terminated explicitly, or implicitly.	⊞	-
49.	Role methods may have various access modifiers.	□	-
50.	Roles may provide meta-functionality.	□	-
51.	Roles allow for typed references.	⊞	Sect. 9.2.2
52.	Roles may be used as filters.	■	Listing 9.6
53.	Roles may be used for renaming.	□	-
54.	Roles may be parameterized.	■	-

Table 11.2.: Summary for the qualitative evaluation of runtime role features supported by *SCROLL*, derived from Graversen [2006]. This summary differentiates between fully (■), partly (⊞), and unsupported (□) features.

28. *Each role type played must be unique.*
SCROLL has an explicit notion for defining how many roles can be played and which role types should be allowed using `RoleGroups` (see Sect. 9.2.9).

29. *Possible supertypes for classes can be class types, role types, or compound object types.*
Since *SCROLL* uses plain Scala classes and case classes as players and roles, this

feature cannot be addressed. One is able to freely mix those supertypes, both within the inheritance and the role-playing relationship, very similar to Gottlob et al. [1996]. Thanks to *SCROLL*'s underlying host language (Scala), support for non-cyclic inheritance hierarchies comes for free.

30. *The amount of simultaneously existing instances of a role type may be constrained.*
    This feature is not supported directly. With *SCROLL*, one may want to use abstract roles instead. Nevertheless, for enforcing singleton roles, Scala's `object` keyword is available. Role binding, removal and transfer has to be stated explicitly.

31. *The amount of players, a role is played by, may be constrained.*
    The amount of players is not constrained in *SCROLL*, which is motivated by practical issues concerning testing and development of roles and players. However the default case is to have one and only one player. Having multiple players for one role instance can be emulated within *SCROLL* easily. The `play`-operator is extensible, allowing for attaching several players to a single role instance. The configurable dispatch (`DispatchDescription`, Sect. 9.3.3) allows for querying the correct, context-dependent behavior if needed, solving signature clashing issues behind the scenes.

32. *The visibility of roles during dispatching may be constrained.*
    Roles in *SCROLL* are public to other entities. Private roles not supported, but may be a target for future development. Currently, protected roles are also not available in *SCROLL*, but may be easily implemented using guarding checks around the calls to methods of protected roles.

33. *Role types are supertypes, subtypes, or unrelated types of their player.*
    Since *SCROLL* has the design goal to bridge the worlds of statically typed host languages and the dynamics of role-playing objects, it is desirable to view roles as unrelated types. Here, dynamic dispatch has to be addressed explicitly, which is done with *SCROLL*'s `DispatchDescription`.

34. *Role types may extend role types, class types, or compound objects.*
    That feature is fully supported in *SCROLL*. It allows for better code reuse. Although a compound object exists at runtime, it can be extended like a class. This requires a special runtime binding, which *SCROLL* offers with its *play*-function. In addition, in *SCROLL* one role type can extend another role type.

35. *The player type for a role type may be a role type, a class type, an interface type, a metaclass, a compound object type, a property, or undefined.*
    The standard case in *SCROLL* is the specification of a class type as player type. Although, there are no *interfaces* in Scala, roles can be attached to subtypes of Java interfaces (as interfaces cannot be instantiated), or to any subtype of Scala traits. Deep roles are fully supported with *SCROLL*. Furthermore, a compound object, created from the players with all of its roles, can be viewed as one object and thus is allowed to play roles within *SCROLL*. A role can be a role for a property (either a field or a method). Since one is able to objectify functions to function objects in Scala, roles can be attached to them. Sadly, having *metaclasses* as players is not applicable for *SCROLL*, since one does not have direct access to Scala's metaclasses (except using reflection or macros).

36. *Properties of roles can be fields, methods, class methods, and static methods.*
    SCROLL allows for selecting the correct field depending on its context to solve

signature clashing problems (`DispatchDescription`, Sect. 9.3.3). The same holds for methods with identical signatures. Hence, methods of roles can override the methods of their player, and vice versa. The developer is free to declare the desired behavior using function composition while traversing the role-play graph as described in Sect. 8.2.3. *SCROLL*, allowing role types to be implemented using classes and case classes, handles static methods as well via companion objects. The role binding of *SCROLL* is not affected by the usage of static methods. Nevertheless, static methods are typically non-virtual (e.g., in Java, C#, Python, and C++) and hence cannot be overridden. The same holds for Scala. Hence, within *SCROLL* a role cannot override methods defined as static methods in a companion object. Since static methods are addressed by names rather than by references, multiple views can only be provided by creating new names and classes containing simple forwarding methods. This, however, requires the change of names for each use expression and is objectionably verbose [Graversen, 2006].

37. *Roles can have nested methods, roles and classes.*
    This feature is supported by *SCROLL* but virtual inner classes are not available in Scala. Implicit role binding via nested roles is not supported. Compartments into which roles are nested, offer the collaboration scope, while the roles are placeholders for the actual participants of the collaboration. As a side-note, one can implement and use nested methods in *SCROLL*, as they are supported by the host language Scala.

38. *Role instances can be referenced directly, or indirectly.*
    With *SCROLL* implementing roles as standard classes or case classes, the desired role instance may be saved and referenced directly at any point during the execution of the program. Hence, indirect references are not supported.

39. *A reference to a role always points to the compound object.*
    This is supported. In *SCROLL*, calling a method on a compound object needs to be resolved using four-dimensional dispatch, as explained in Sect. 9.3.3, since finding the correct behavior may be ambiguous and / or context-dependent.

40. *Method dispatch on roles happens on the sender or its player, the receiver or its player, the context, or the compound object.*
    This is supported by *SCROLL* using its `DispatchDescription` concept as explained in Sect. 9.3.3. The method lookup on the compound object is performed by investigating all roles, descendant roles (deep roles), and the player itself. Problems like self-recursion and ambiguities when a message could be answered by different roles, are resolved at runtime. This leads to a very flexible understanding of role dispatch within *SCROLL*.

41. *Self may refer to dual self, or non-virtual self.*
    Dual self refers to the initial receiver (the compound object) and the currently executing part object (a role instance). In *SCROLL*, one may access the first one with the +-operator (lifting, see Sect. 8.2.2), and the latter through a normal reference to the desired object (e.g., using `this` or `super`, accessing the static inheritance hierarchy). Because *SCROLL* implements roles and players as unrelated entities, the concept of non-virtual self fits very naturally as it resembles the definition of components, but also enables roles to be applied to several unrelated players. Hence, non-virtual self is supported, as well.

42. *Super refers both to the static inheritance chain, and to the attached roles.*
SCROLL does not implement a role-playing-specific super call, only Scala's standard `super` can be used for traversing the static inheritance hierarchy. But with the +-operator and its implicitly created compound object (see Sect. 8.2.2), one has access to the aggregated behavior of all roles when relying on the standard super call is not sufficient.

43. *The player may be referenced directly, or indirectly.*
With the reference to a *SCROLL* compound object, generated with the +-operator, i.e., an indirect player reference, roles can be aggregated, although they are physically not attached to each other. The mapping is maintained behind the scenes, as explained in Sect. 8.2.3. Hence, the player can be referenced indirectly. Direct references are supported as well, since players are instances of standard Scala classes or case classes.

44. *A role may be called from its player.*
This feature is supported via the +-operator and its implicitly created compound object (see Sect. 8.2.2).

45. *Roles may call among each other.*
SCROLL implements full reification of roles, referencing and invoking methods on them with its concept of implicit conversions (see Sect. 8.2.2) and compiler rewrites (see Sect. 8.2.1).

46. *Roles may incorporate around-methods.*
SCROLL does not incorporate the feature of around-methods as first-class citizen.

47. *Role creation, attachment, and movement may be restricted.*
SCROLL utilizes a fairly simple role creation technique. In the first place, there are no role creation constraints, no object pooling for role caching, and roles cannot be bound implicitly. Nevertheless, for a fine-grained role binding semantics, the `RolePlayingAutomaton` (see Sect. 9.2.10) may be used. As roles are implemented with standard classes or case classes in *SCROLL*, they may float around without a player. The conjunctive attachment strategy to build the compound object is used in *SCROLL*. The disjunctive attachment of roles is supported by *SCROLL* with a combination of: i) its specific notion of identity (see Sect. 8.2.2), and ii) when asking the compound object or a part object to return a role of a specific type, the request is sent to the compound object, utilizing a full transitive traversal of all roles using the role-play graph (see Sect. 8.2.3). *SCROLL* does not restrict the player's type for roles, hence a role can be freely moved around on instances. On the downside, there are no mechanics implemented in *SCROLL* (like blocking mutexes or semaphores) to prevent executing role-specific behavior in the wrong context. Furthermore, contracts can only be checked in between individual role movements. Automatic checking is not supported. The role constraints presented in Riehle [2000] are fully incorporated within *SCROLL*, in addition to its `RoleGroups` (see. Sect. 9.2.9). Those allow for constraining the relationships between roles and role movement.

48. *Roles may be terminated explicitly, or implicitly.*
Explicit role termination is not supported in *SCROLL*. As the DSL is implemented upon the JVM, it solely relies on the garbage collector. Since the access to the call stack is impossible within Scala and explicit memory management is not applicable to the JVM in general, *SCROLL* does not support the feature of explicit role removal.

49. *Role methods may have various access modifiers.*
In *SCROLL*, roles are represented as unrelated types to their player for a maximum of flexibility during development and testing. The dispatch target is accessed via reflection (as explained in Sect. 8.2.1). All access modifiers are ignored. Hence, the role can access and override properties and behavior of their players by default. Roles in *SCROLL* can re-define an otherwise final method of its player. Since all kinds of exceptions are collected and wrapped like explained in Sect. 8.2.1, roles in *SCROLL* cannot introduce new exceptions.

50. *Roles may provide meta-functionality.*
SCROLL does not support this feature.

51. *Roles allow for typed references.*
While roles are implemented as separate types within *SCROLL*, the player type is assigned to the least specific type in the type system of Scala, which is `Any`. The disadvantage is that the role cannot rely on any other properties of its player than those defined in that type. Type casts are unsafe at that point. However, runtime checks are provided, when roles are attached to their players to ensure that signatures of role- and player-specific attributes and behavior have the correct types. Dispatch in *SCROLL* is based upon the dynamic type of the compound object. This makes the dispatch for compound objects highly flexible. The set of roles aggregated is checked for all messages arriving during dispatching.

52. *Roles may be used as filters.*
This feature is supported, as it maps very naturally to *SCROLL*'s point of view. Roles in *SCROLL* do not rely on any property of their player, because they have no direct relationship to them, as they are modeled as unrelated types. Filtering messages is type-safe and happens when using consultation during the process of dispatching.

53. *Roles may be used for renaming.*
SCROLL does not require this feature, since ambiguities due to signature clashing are resolved using the concept of a `DispatchDescription` (see Sect. 9.3.3).

54. *Roles may be parameterized.*
SCROLL uses standard classes and case classes for the specification of roles and players. With Java's and Scala's generic types, the feature of parameterized roles is fully incorporated.

This section investigated the semantics of roles by investigating the runtime features presented in Sect. 3.7. It was shown how *SCROLL* supports those features. *Class instances* as the fundamental basis of the objects semantics of *SCROLL* with their corresponding role-playing constraints and supertype restrictions are fully incorporated. Furthermore, many *constraints* with regard to the cardinalities imposed on the player as well as the role side are supported. *Relationships*, e.g., with the concept of inheritance, can be handled and most of the well-known *properties* (e.g., static methods, class methods and fields) are available within *SCROLL*. In addition, the analysis for role-specific *behavior* reveals *SCROLL*'s ability to dispatch calls on various entities. For the *life cycle* of roles, *SCROLL* offers support for a fairly simple implementation of role creation, attachment, movement, and removal. A compact summary can be found in Table 11.2. Additionally, references to code examples throughout the thesis are given.

11.2. QUANTITATIVE EVALUATION

In this section, the *SCROLL* library is quantitatively evaluated by employing the real-world scenario of a small banking application, extracted from Reenskaug and Coplien [2009]. A bank is a financial institution providing banking services to their customers, who are persons. Customers may be advised by consultants. They own savings and checking accounts, and perform transactions. Transactions encapsulate the process of transferring money from exactly one source account to one target account. The customer may initiate them, however, each transaction is managed and executed by the bank. Additionally, financial regulations require that no account can be both a checking and a savings account, as well as both the source and target of the same transaction.

11.2.1. THE MODEL

With the graphical model presented in Fig. 11.1, the following CROM can be derived:

Definition 9 (Compartment Role Object Model for the Bank Example)
Let $\mathscr{B} = (NT, RT, CT, RST, \text{fills}, \text{parts}, \text{rel})$ be the model of the bank, where the individual components are defined as follows:

$$
\begin{aligned}
NT &:= \{\text{Person, Account}\} \\
RT &:= \{\text{Customer, CA, SA,} \\
&\qquad \text{Source, Target, MoneyTransfer}\} \\
CT &:= \{\text{Bank, Transaction}\} \\
RST &:= \{\text{own_ca, own_sa, trans}\} \\
\mathit{fills} &:= \{(\text{Person, Customer}), \\
&\qquad (\text{Account, Source}), \\
&\qquad (\text{Account, Target}), (\text{Account, CA}), \\
&\qquad (\text{Account, SA}), \\
&\qquad (\text{Transaction, MoneyTransfer})\}
\end{aligned}
\qquad
\begin{aligned}
\mathit{parts} &:= \{\text{Bank} \to \{\text{Customer,} \\
&\qquad \text{CA, SA, MoneyTransfer}\}, \\
&\qquad \text{Transaction} \to \{\text{Source, Target}\}\} \\
\mathit{rel} &:= \{\text{own_ca} \to (\text{Customer, CA}), \\
&\qquad \text{own_sa} \to (\text{Customer, SA}), \\
&\qquad \text{trans} \to (\text{Source, Target})\}
\end{aligned}
$$

The bank model $\mathscr{B}$ is created in four steps. First, all the natural types, compartment types, role types, and relationship types are specified. Secondly, the set of role types is assigned to a compartment type with the parts-function. Thirdly, it is specified which natural type can fill which role type, and finally the rel-function is defined for the role types at the ends of a relationship type. On instance level, we distinguish naturals, roles, compartments and links as instances of their respective types. The following CROI is a valid instance of the CROM $\mathscr{B}$:

Definition 10 (Compartment Role Object Instance for the Bank Example)
Let $\mathscr{B} = (NT, RT, CT, RST, \mathit{fills}, \mathit{parts}, \mathit{rel})$ be the well-formed CROM for the bank example as defined above; then $\mathfrak{b} = (N, R, C, \mathit{type}, \mathit{plays}, \mathit{links})$ is an instance of that model, where the components are defined as follows:

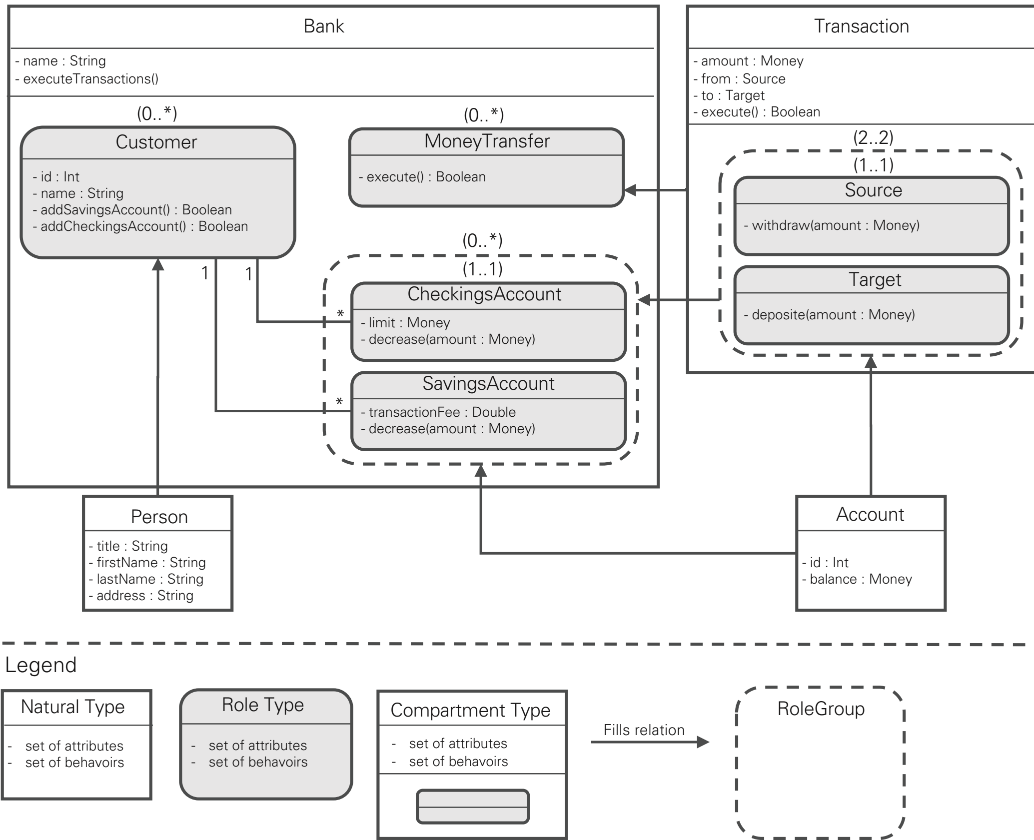


Figure 11.1.: The bank example model in CROM notation.

$$
\begin{aligned}
N &:= \{\text{Peter},\ \text{Klaus},\\
&\qquad \text{Account}_1,\ \text{Account}_2\}\\
R &:= \{\text{Cu}_1,\ \text{Cu}_2,\ \text{Ca},\ \text{Sa},\ \text{S},\ \text{T},\ \text{M}\}\\
C &:= \{\text{bank},\ \text{t}\}\\
\mathit{type} &:= \{(\text{Cu}_1 \to \text{Customer}),\\
&\qquad (\text{Cu}_2 \to \text{Customer}),\\
&\qquad (\text{Ca} \to \text{CA}), (\text{Sa} \to \text{SA}),\\
&\qquad (\text{S} \to \text{Source}), (\text{T} \to \text{Target}),\\
&\qquad (\text{M} \to \text{MoneyTransfer}),\\
&\qquad (\text{bank} \to \text{Bank}),\\
&\qquad (\text{t} \to \text{Transaction}), \dots\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{plays} &:= \{(\text{Klaus},\ \text{bank},\ \text{Cu}_1), (\text{Peter},\ \text{bank},\ \text{Cu}_2),\\
&\qquad (\text{Account}_1,\ \text{bank},\ \text{Sa}),\\
&\qquad (\text{Account}_2,\ \text{bank},\ \text{Ca}), (\text{t},\ \text{bank},\ \text{M}),\\
&\qquad (\text{Account}_1,\ \text{t},\ \text{S}), (\text{Account}_2,\ \text{t},\ \text{T})\}\\
\mathit{links} &:= \{(\text{own_ca},\ \text{bank}) \to \{(\text{Cu}_1,\ \text{Ca}), (\text{Cu}_2,\ \varepsilon)\},\\
&\qquad (\text{own_sa},\ \text{bank}) \to \{(\text{Cu}_1,\ \varepsilon), (\text{Cu}_2,\ \text{Sa})\},\\
&\qquad (\text{trans},\ \text{t}) \to \{(\text{S},\ \text{T})\}\}
\end{aligned}
$$

The CROI $\mathfrak{b}$ is created by collecting all the naturals, compartments, and roles accordingly; mapping their respective types; linking the roles to their players; and assigning a tuple for each depicted relationship. $\mathfrak{b}$ must contain a tuple for the roles Cu_1 and Cu_2 in the *own_ca*, *own_sa*, and *advises* relationships regardless of their actual relation to a counter role. These tuples link those roles to the empty counter role ε instead.

Finally, the constraining role groups and the constraint model itself can be defined on top of the previous definitions. Role groups constrain the set of roles an object o is allowed to play simultaneously in a certain compartment c. In case a is a role type, $rt^{\mathcal{I}_o^c}$ checks whether o plays a role of type rt in c. If a is a role group (B, n, m), it should check

Implementation	Listing	Implementation	Listing
SCROLL	D.25 (page 238)	SingleType	D.27 (page 244)
ScalaRoles	D.24(page 235)	SubtypeHiddenDel.	D.28 (page 246)
OT/J	D.32 (page 257)	SubtypeInternalFlag	D.29 (page 249)
Role Object Pattern	D.20 (page 231)	SubtypeStateObject	D.30 (page 252)
SeparateType	D.26 (page 241)		

Table 11.3.: Evaluation implementations and references to their full code listings.

whether the sum of the evaluations for all $b \in B$ is between n and m. In general, both the role constraints from Riehle [2000] and any propositional formula can be represented with role groups. For instance, a role-implied constraint from consultant to customer would be modeled as: $(\{(\{Consultant\},0,0),Customer\},1,2)$. This is equivalent to the formula $\neg Consultant \vee Customer$ and fulfills the intended semantics of the role-implied constraint.

Definition 11 (Role Groups and Constraint Model for the Bank Example)
Let $\mathscr{B}$ be the bank model from Fig. 11.1. Then $\mathscr{C}_{\mathscr{B}} = (rolec, card, intra)$ is the constraint model where the components are defined as:

$$
\begin{aligned}
rolec &:= \{(0..\infty,\ \text{bankaccounts}),\\
&\qquad \text{Transaction} \to \{(2..2,\ \text{participants})\}\}\\
card &:= \{\text{own_ca} \to (1..\infty,\ 0..\infty), \qquad\qquad bankaccounts := (\{\text{CA},\ \text{SA}\},1,1)\\
&\qquad \text{own_sa} \to (1..\infty,\ 0..\infty), \qquad\qquad participants := (\{\text{Source},\ \text{Target}\},1,1)\\
&\qquad \text{trans} \to (1..1,\ 1..1)\}\\
intra &:= \{(\text{advises},\ \text{irreflexive})\}
\end{aligned}
$$

The constraint model can be obtained by mapping the graphical constraints to their formal counterparts: role groups with cardinalities to the *rolec*-mapping, relationship cardinality to the *card*-function, and intra-relationship constraints to the *intra*-relation.

Exactly two distinct accounts must fulfill the *participants* role group in the *transaction* compartment. Because Peter is a customer in the $bank$, $Account_1$ and $Account_2$ are the respective source and target in the transaction t, this is fulfilled. Additionally, each object playing a role in a compartment must fulfill those role groups containing the corresponding role type. In $\mathfrak{b}$, all accounts individually satisfy the *participants* and *bankaccounts* role group. Furthermore, the number of *successors* for the first place and *predecessors* for the second place for each link (relationship instance) is computed and checked against the limits imposed by the cardinality constraints. In case of $\mathfrak{b}$, the number of successors and predecessors ranges from zero (for Cu_1 in the *own_sa* relationship and Cu_2 in the *own_ca* relationship) to one (for all other roles and relationships).

11.2.2. THE IMPLEMENTATION

The bank example introduced in the previous section has been implemented with various role-base languages and implementation patterns for roles. Multiple dimensions of scaling are explored for gaining insight into the performance and scaling characteristics of those approaches. We varied the number of players, roles, and transactions in the system as shown in Table 11.4. In the largest benchmark, 1000 transaction compartments (each with 1000 persons playing the customer role 1000 times, and each having a savings account attached) are active. Attributes as well as the selection of source and target

Parameter	Variations	Meaning
Player	10, 100, 1000	Number of persons having accounts in the bank.
Role	10, 100, 1000	Number of customers in the bank. Each player plays the role of a customer multiple times, each time having exactly one savings account attached to it.
Transaction	10, 100, 1000	Number of transactions in the bank. In each transaction, two randomly selected accounts play the role of the source and target account, respectively. Additionally, that transaction plays the role `MoneyTransfer` doing the actual transfer of money between those two accounts.

Table 11.4.: Variation of parameters for the bank example benchmark.

accounts for the actual transactions were generated and selected randomly. The full implementations are available as shown in Table 11.3. As an example, we explain the code for the implementation of the bank model with *SCROLL* and OT/J, respectively.

SCROLL The natural types of the example are implemented as standard Scala case classes as follows:

```
case class Person(title: String, firstName: String, lastName: String,
    address: String)
case class Account(var balance: Money, id: Integer) {
  def increase(amount: Money): Unit = {
    balance = balance + amount
  }
  def decrease(amount: Money): Unit = {
    balance = balance - amount
  }
}
```

The separately defined `Transaction` compartment handling the actual money transfer is specified as Scala trait containing the roles `Source` and `Target` decreasing (Line 12) and increasing (Line 16) the respective balance of their underlying players (`Account`) by accessing that behavior via the +-operator (as explained in Sect. 9.2.2):

```
trait Transaction extends Compartment {
  var amount: Money = _
  var from: Source = _
  var to: Target = _
  def execute(): Boolean = {
    from.withdraw(amount)
    to.deposite(amount)
    true
  }

  case class Source() {
    def withdraw(amount: Money): Unit = { +this decrease amount }
  }

  case class Target() {
    def deposite(amount: Money): Unit = { +this increase amount }
  }
}
```

Based on this, the `Bank` compartment (defined as Scala trait as well) executes all

attached transactions by simply delegating them all to the MoneyTransfer role (see Line 3):

```
trait Bank extends Compartment {
  val moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]
  def executeTransactions(): Unit = { moneyTransfers.foreach(_.execute()) }
  /* ... */
}
```

The Customer role is specified inside the Bank compartment. Accounts can be added and the respective play-relation is defined for specific types of accounts as roles (SavingsAccount in Line 5, CheckingsAccount in Line 9):

```
case class Customer(name: String, id: Integer) {
  val accounts = mutable.ArrayBuffer.empty[Account]
  def addSavingsAccount(a: Account): Boolean = {
    accounts.append(a)
    a play SavingsAccount(0.1); true
  }
  def addCheckingsAccount(a: Account): Boolean = {
    accounts.append(a)
    a play CheckingsAccount(Money(100, "USD")); true
  }
}
```

The role MoneyTransfer is also defined inside the Bank compartment and delegates its work (execute) to its underlying player, a Transaction instance (see Line 4). As both implement the same behavior, a signature clash appears (execute in MoneyTransfer and execute in Transaction). This is resolved by providing an explicit dispatch description simply bypassing instances of MoneyTransfer. Hence, it is ensured that only the player instance itself will receive the call to the desired behavior (see Line 3) as this call is delegated backward to the core.

```
case class MoneyTransfer() {
  def execute(): Boolean = {
    implicit val dd = Bypassing(_.isInstanceOf[MoneyTransfer])
    +this execute()
  }
}
```

The same holds for the implementation of the roles SavingsAccount and CheckingsAccount. Additionally, those roles alter the core behavior by applying the respective business rules on-top:

```
case class CheckingsAccount(var limit: Money) {
  def decrease(amount: Money): Unit = amount match {
    case a if a <= limit =>
       implicit val dd = Bypassing(_.isInstanceOf[CheckingsAccount])
       +this decrease amount
    case _ => throw new IllegalArgumentException("Amount > limit!")
  }
}
case class SavingsAccount(var transactionFee: Double) {
   def decrease(amount: Money): Unit = {
     implicit val dd = Bypassing(_.isInstanceOf[SavingsAccount])
     +this decrease (amount + amount * transactionFee)
   }
}
```

Finally, the definitions for the role groups can be added:

```
RoleGroup("transaction").containing[Source, Target](1, 1)(2, 2)
RoleGroup("accounts").containing[CheckingsAccount, SavingsAccount](1,
  1)(0, *)
```

Hardware Platform
macOS Sierra 10.12.1
1,7 GHz Intel Core i7
8GB 1600 MHz DDR3
Java Platform
Java Version 1.8.0_112
Java™ SE Runtime Environment (build 1.8.0_112-b16)
Java HotSpot™ 64-Bit Server VM (build 25.112-b16, mixed mode)
Scala Platform
Scala Code Runner Version 2.12.0
Scala Compiler Version 2.12.0
OT/J and Eclipse Platform
org.eclipse.objectteams.runtime_2.5.0.201606070956.jar
org.objectweb.asm_5.0.1.v201404251740.jar
org.objectweb.asm.commons_5.0.1.v201404251740.jar
org.objectweb.asm.tree_5.0.1.v201404251740.jar
org.objectweb.asm.commons_5.0.1.v201404251740.jar
ecotj-R-2.5.0-201606070953.jar
Procedure
10 global JVM warmup runs
10 JVM warmup runs before each individual benchmark run
10 repetitions for each individual benchmark run
Median calculation over benchmark runs

Table 11.5.: The benchmark environment.

ObjectTeams/Java The implementation in OT/J is very similar (see Listing D.32). Naturals are defined via standard Java static classes. All role types are defined inside `team` classes as normal classes. In contrast to *SCROLL*'s +-operator, accessing the core player behavior is done via `callin` bindings and `base` as follows, e.g., for the `Source` role:

```
1 public class Source playedBy Account {
2    callin void withdraw(double amount) { base.withdraw(amount); }
3    withdraw <- replace decrease;
4 }
```

11.2.3. THE EVALUATION METHOD

The quantitative evaluation benchmarks the individual implementations of the bank example to gain insight into the respective performance characteristics. A JVM-based benchmark platform was implemented (see Table 11.5).

Two aspects were measured. First, the bank example is instantiated for each individual implementation, instantiating all player, role and compartment types. Additionally, required associations and play-relations are built. These activities are measured as *build times*. Secondly, the actual money transactions are performed resulting in the overall *execution times*.

On the technical side, all benchmarks were performed with JVM pre-warming and multiple runs to automatically eliminate noise due to JIT compilation, garbage collection or undesired heap allocation patterns. There are many mechanisms in the JVM which are

transparent to the programmer, for instance, automatic memory management, dynamic compilation and adaptive optimization. Importantly, these mechanisms are triggered implicitly by the code being executed, and the programmer has little or no control over them.

JIT compilation The HotSpot compiler continually analyzes the program performance for parts of the program executed frequently and compiles those parts down to machine code. Any part of the code can potentially be chosen for compilation at any point during the runtime, and this decision may be taken in the midst of running a benchmark, yielding an inaccurate measurement. Also, portions of the program are periodically recompiled but can also be de-optimized based on the JVM runtime information. Hence, during the runtime of the program, the same code might exhibit very different performance characteristics.

Classloading Since the JVM, unlike a typical compiler, has global program information available, it can apply non-local optimizations. This means that a method may be optimized based on the information in some seemingly unrelated method. One such example is inline caching, where the JVM can inline polymorphic method calls. Since not all the classes loaded in the complete application are loaded in a benchmark, many of the callsites in a benchmark can and will be optimized, thus yielding an inaccurate running time measurement. As a result, seemingly unrelated code can have a big impact on performance of the code being benchmarked.

Automatic memory management The benchmark is simply a piece of code running and measuring the running time of some other piece of code. As such, it may inadvertently leave the memory heap in a state which affects subsequent allocations or trigger a garbage collection cycle, each of which changes the observed performance of the code being tested. In a real-world application, the state of the heap is unpredictable, and in general different heap states and allocation patterns tend to give very different performance results.

There are other non-JVM considerations to take into account as well. Specific processors, cache and memory sizes may show a very different performance for the same benchmark. On a single processor system, concurrently running applications or operating system events can cause a degradation in performance. Different Java Runtime Environment (JRE) versions may yield entirely different results. These effects do not cause a linear degradation in performance. However, decreasing the memory size twice may cause the benchmark to run a hundred times slower. Additionally, performance of some code is not some absolute scalar value denoting how fast the code is. Rather, it is some function which maps the inputs and the runtime conditions to a running time. This function is impossible to reproduce analytically.

In conclusion, benchmark conditions and inputs are hard to reproduce. Performance measurement in general introduces observer bias and runtime behavior is in general non-deterministic. With that, performance metrics inherently give an incomplete picture about the performance characteristics. With that being said, this benchmark portrays neither the real-world behavior of role-based applications, nor does it result in a precise, reproducible performance measurement. Still, the benchmarking of the individual role implementations is important because it gives insights about the performance characteristics in some particular conditions. This information might not be complete, but it captures some characteristics that future role researchers might be interested in.

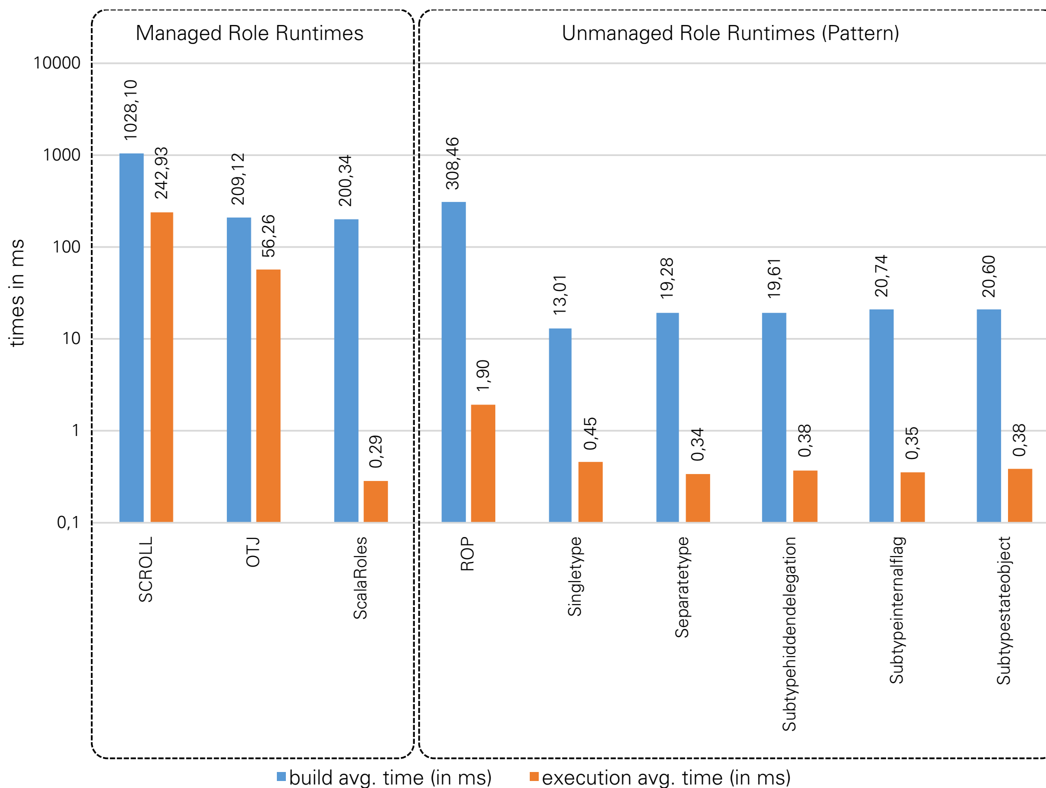


Figure 11.2.: Overall execution and build times of the bank example.

11.2.4. RESULTS AND SUMMARY

Even if the implemented benchmark based on the bank example does not reflect the real-world behavior of role-based applications, it gives basic insights about the performance characteristics of *SCROLL* when scaling the number of player, role and compartment instances and the respective number of associations between them. In the largest benchmarking case (with 1000 transaction compartments, each with 1000 persons playing the customer role 1000 times, and each having a savings account attached), *SCROLL* performs roughly twenty times slower than OT/J, and about four times slower than ScalaRoles or the hand-crafted implementation when considering the build times (see Fig. 11.3). The overall performance disadvantage is at around the factor of ten (see Fig. 11.4). Statically modeled and compiled version of the bank example (e.g., with separate types or hidden delegation) are faster, but lack almost all of the features of role-based programming. For these implementation patterns, the whole adaptation logic, true delegation and the semantics of roles need to be manually implemented and managed.

Overall, calculating the average values over build and execution times respectively, *SCROLL* performs roughly five times slower than OT/J and ScalaRoles (see Fig. 11.2). Again, the manually managed implementations with patterns are faster. The reason behind this is *SCROLL*'s use of the Java Reflection API to gather and manipulate the behavior and structure at runtime. Via reflection, performing such tasks is, by magnitudes, more expensive. Because reflection involves types that must be dynamically resolved, most of the JVM optimizations cannot be applied. Consequently, reflective operations should be avoided in sections of code which are called frequently in performance-sensitive applications. The focus of the thesis is on developing new methods for 4-dimensional dispatch for role-based programming, but not on optimization as such, which is considered to be the second step.

	Transactions									Implementation
	10			100			1000			
roles / players	10	100	1000	10	100	1000	10	100	1000	
10	76,59	66,88	54,89	14,23	12,22	50,25	85,36	89,71	243,26	SCROLL
100	94,18	93,43	535,33	15,76	54,90	2237,80	88,39	1829,41	700,85	
1000	224,25	513,96	6090,02	45,61	496,43	5288,26	123,29	716,99	7916,31	
10	1,48	5,78	19,90	1,15	2,42	25,88	4,63	5,50	21,99	ScalaRoles
100	10,46	32,22	162,36	2,40	14,57	191,47	5,89	17,33	184,07	
1000	61,85	163,96	1391,93	16,74	135,37	1360,65	20,85	157,68	1390,74	
10	0,68	2,72	25,50	0,48	2,61	22,80	1,50	3,12	21,53	ROP
100	3,60	20,49	199,43	2,24	20,27	204,39	3,44	20,73	211,60	
1000	25,38	213,34	2305,54	23,04	209,84	2283,61	24,54	229,06	2246,82	
10	0,56	1,57	5,56	0,63	0,64	2,74	1,50	1,31	5,37	Singletype
100	3,87	2,57	7,71	0,57	1,76	12,33	1,18	1,96	15,22	
1000	2,91	8,74	72,15	1,43	8,18	106,16	2,21	9,56	72,89	
10	0,67	1,46	3,32	0,43	0,63	3,94	1,07	1,34	4,99	Separatetype
100	1,52	5,00	16,97	0,49	1,98	19,25	1,38	2,64	22,31	
1000	5,86	18,52	129,54	1,91	15,31	149,51	3,07	17,44	89,97	
10	0,68	1,30	4,63	0,46	0,55	4,69	1,24	1,58	4,94	Subtypehiddendelegation
100	1,07	3,83	11,67	0,58	2,09	18,20	1,38	2,60	24,03	
1000	3,51	8,21	126,09	1,90	16,90	164,40	2,90	18,56	101,54	
10	0,62	1,39	5,52	0,37	0,53	3,55	1,07	1,60	5,71	Subtypeinternalflag
100	1,03	4,24	12,63	0,50	2,01	17,72	1,33	4,13	18,34	
1000	3,55	10,69	127,63	2,00	14,80	194,23	2,98	19,88	101,87	
10	0,66	1,66	4,53	0,45	0,82	4,22	1,10	1,28	5,55	Subtypestateobject
100	1,28	5,52	24,51	0,73	2,94	18,96	1,23	2,70	17,94	
1000	5,18	16,40	156,21	2,78	19,66	141,23	2,84	19,81	95,98	
10	161,64	183,15	190,73	162,28	167,19	190,44	167,89	174,13	197,86	OTJ
100	173,18	187,87	207,79	173,94	174,30	220,57	172,87	185,74	241,12	
1000	187,59	236,52	349,50	171,92	212,75	360,62	191,75	243,41	359,39	

Figure 11.3.: Heatmap for build times of the bank example.

	Transactions									Implementation
	10			100			1000			
roles / players	10	100	1000	10	100	1000	10	100	1000	
10	1,80	9,33	62,88	79,03	36,54	215,07	387,11	593,56	1063,67	SCROLL
100	12,02	14,08	62,35	67,39	26,40	67,23	511,96	519,13	1065,49	
1000	16,22	12,26	34,91	18,74	19,19	66,32	362,63	349,55	884,37	
10	0,01	0,07	0,56	0,01	0,06	0,64	0,01	0,08	0,69	ScalaRoles
100	0,01	0,07	0,78	0,01	0,05	0,79	0,01	0,08	0,67	
1000	0,01	0,10	0,88	0,01	0,08	0,86	0,01	0,08	1,13	
10	0,03	0,25	1,45	0,04	0,34	3,00	0,07	5,83	6,44	ROP
100	0,03	0,16	3,04	0,03	0,28	4,18	0,16	0,65	8,54	
1000	0,03	0,18	1,19	0,05	0,32	1,77	0,16	1,39	11,81	
10	0,02	0,16	0,84	0,02	0,16	0,75	0,02	0,21	0,78	Singletype
100	0,02	0,11	1,59	0,02	0,12	1,50	0,02	0,14	1,23	
1000	0,02	0,15	1,44	0,04	0,15	1,30	0,02	0,16	1,20	
10	0,01	0,12	0,74	0,02	0,13	0,55	0,03	0,18	0,80	Separatetype
100	0,02	0,09	1,30	0,01	0,08	1,10	0,02	0,09	0,80	
1000	0,02	0,09	0,77	0,02	0,10	0,94	0,01	0,10	1,18	
10	0,01	0,13	0,84	0,02	0,11	1,23	0,02	0,13	0,80	Subtypehiddendelegation
100	0,01	0,09	1,42	0,03	0,06	1,17	0,01	0,05	0,76	
1000	0,02	0,11	0,80	0,02	0,08	0,91	0,01	0,10	1,24	
10	0,02	0,15	0,73	0,01	0,17	0,93	0,01	0,13	0,46	Subtypeinternalflag
100	0,02	0,08	1,21	0,01	0,08	1,12	0,02	0,10	0,75	
1000	0,02	0,14	1,04	0,03	0,09	1,14	0,01	0,10	1,00	
10	0,02	0,11	0,80	0,02	0,09	1,16	0,01	0,12	0,70	Subtypestateobject
100	0,02	0,09	1,51	0,02	0,09	1,41	0,03	0,07	0,81	
1000	0,02	0,16	0,88	0,02	0,13	0,75	0,03	0,17	1,14	
10	17,63	19,38	18,87	37,13	36,61	41,30	100,78	105,94	108,99	OTJ
100	15,52	20,99	18,48	36,01	45,28	37,72	94,63	103,77	116,62	
1000	18,54	13,76	17,17	43,22	39,06	34,87	100,34	120,21	156,14	

Figure 11.4.: Heatmap for execution times of the bank example.

11.3. DISCUSSION

The evaluation for *SCROLL* in the scope of this thesis was split into four parts. First, we analyzed the fulfillment of the requirements stated in Sect. 7. Secondly, *SCROLL* was analyzed based on a previously defined classification scheme [Kühn et al., 2014]. Then, the variability analysis from Graversen [2006] was applied to *SCROLL*. Finally, we benchmarked various implementations for roles at runtime and identified performance bottlenecks of *SCROLL*. In sum, these are the results:

SCROLL is a very general approach The evaluation with regard to the derived requirements clearly shows the various advantages of *SCROLL*, as it is able to implement all of them, only failing at one (the required performance as explained in Sect. 11.2).

Feature-based analysis of SCROLL To investigate how well the implementation with *SCROLL* blends into contemporary approaches, the previously defined scheme from Kühn et al. [2014] with 26 classifying features of roles was applied. *SCROLL* fully implements 22 of them.

Summary for runtime feature analysis This section investigated the role semantics by a feature analysis loosely based on Graversen [2006]. It was shown how *SCROLL* supports those features. Instances of *classes* as the fundamental basis of roles in *SCROLL* with their corresponding role-playing constraints and supertype restrictions are fully incorporated. Furthermore, many *constraints* with regard to the cardinalities imposed on the player as well as the role side are supported. *Relationships*, e.g., with the concept of inheritance, can be handled and most of the well-known *properties* (e.g., static methods, class methods and fields) are available within *SCROLL*. In addition, the analysis for role-specific *behavior* reveals *SCROLL*'s ability to dispatch calls on various entities (e.g., roles and its players, the notion of self, and super). The notion of *identity* is discussed with the question in mind if roles have an unique identity or it is rather shared between a role and its player. When it comes to handling the *life cycle* of roles, *SCROLL* offers support for a fairly simple implementation of role creation, attachment, movement and removal. Finally, *type* related issues are discussed. As a result, it was shown that *SCROLL* realizes a good balance for the role and compartment concepts with regard to statically and dynamically languages.

Summary for quantitative evaluation For our benchmark suite, *SCROLL* performs roughly five times slower than OT/J and ScalaRoles. Manually managed implementations with patterns are way faster. This slowdown stems from the heavy use of the Java Reflection API to gather and manipulate the behavior and structure at runtime. Via reflection, performing such tasks is expensive. Consequently, with reflective operations being much slower than their non-reflective counterparts, they should be avoided in sections of code which are called frequently in performance-sensitive applications. In the scope of this thesis that is not to be considered critical, as it focuses more on the conceptual features of dynamic dispatch.

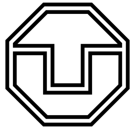
12

THE ADVANTAGES OF *SCROLL*

In the modern software world, software systems are expected to adapt to a changing environment. During their lifetime, new features are requested and existing requirements change. Software written for a specific purpose may become useful in situations and environments, which the developer did not anticipate. Object-oriented programming is widely used to build extensible and flexible software systems. It is successful because it supports programming with data structures that closely resemble the problem domain. However, future software systems expect a higher level of dynamism, which is not offered by classic object-oriented concepts. With dynamically typed, object-oriented scripting languages, a flexible programming style is available. Modules, classes and objects can be extended at runtime. But programming in a dynamically typed language comes at a cost: without static type information, it is not possible to analyze programs statically and catch entire classes of programming errors before actually running the program. The burden is carried solely by the programmer. To cope with challenges posed from ubiquitous and adaptive software systems, research proposed several approaches, including the concept of roles. They allow to extract the context-dependent behavior from the objects itself and model it in separate role types. Together with role-based dispatch, a new level of separation of concerns within those objects is reached. The core behavior and structure of an object is defined in its natural type. Context-dependent and evolving parts are specified in role types. Moreover, role-playing objects are able to start and stop playing roles to adapt their behavior and structure dynamically during runtime, without the need for re-instantiation.

Because a declarative and parameterizable approach for four-dimensional, context-aware dispatch at runtime is not yet available, the method-call interception DSL *SCROLL* and its MOP were presented. The DSL *SCROLL* embeds the dynamic semantics of roles and their dispatch in a statically typed, object-oriented language (Scala), utilizing only those features that are available through its standard compiler. The *SCROLL* library allows for easy integration of legacy code and a high separation of concerns. Having a statically-typed host language for roles supports the developer with the advantages of static typing and dynamic objects at the same time. With *SCROLL*, arbitrary objects can be augmented dynamically with new functionality or state. Moreover, obstacles arising from split-objects can be solved with a compound object, enabled by dynamic conversions and an adapted notion of object identity. Using Scala's `Dynamic` trait together with a definition table allows for easy querying for behavior that is not natively available at the player. For that, *SCROLL* requires a concept for explicitly triggering *compiler rewrites* handing over calls to the library for finding behavior that is not natively available at the player but at its roles (role dispatch). For aggregating a compound object from the player and its roles, *implicit conversions* are needed. The relationships between each player and its roles are stored within the *definition table*, the role-play graph. Additionally, a declarative and parameterizable approach for four-dimensional, context-aware dispatch was presented in *SCROLL* 's MOP that enables the developer of adaptive systems to overcome the ambiguities introduced with role-playing objects. We have demonstrated how objects can be augmented dynamically with new functionality or state grouped together in roles and structured contexts.

If one is able to find or emulate these three concepts in a statically-typed, object-oriented language, it is easy to provide an alternative implementation of *SCROLL*. Hence, the approach is generally applicable.



Faculty of Computer Science Institute of Software- and Multimedia-Technology

Software Technology Group

PART IV.

RELATED WORK, CONCLUSION, AND OUTLOOK

13

ROLES WITH PATTERNS OR OTHER PROGRAMMING LANGUAGES

13.1. ROLES WITH PATTERNS

The following section demonstrates the advantages of the proposed library approach for pure embedding of roles by comparing it to simple, manually instantiated implementations and design patterns widely used when people try to cope with the required dynamics [Fowler, 1997]. Those pattern-based solutions suffer under various problems, such as single, too complex types, restricted scalability, or the need for hand-written management code, as listed in Table 13.1.

The most basic solution would be to use one **single type** for the player and its roles. If they do not differentiate in behavior too much and are not target for future changes, this would be a valid solution without any over-engineering. On the downside, this single type can be complex and, consequently, is hard to maintain. If roles introduce different features, they might be implemented as **separate types**. This in turn, removes coupling and unnecessary tangling of relationships between the types. On the downside, having roles as completely separate types introduces the duplication of features and a loss of integrity with shared state and behavior. Using **subtypes** and putting the common things into the supertype for each role overcomes the issues of the duplication of features and of the loss of integrity with shared state and behavior while still being conceptually simple. On the downside, the resulting inheritance hierarchy is hard to evolve with multiple or changing roles as each of them requires the interface of the supertype to be changed, as well. In general, the classification of domain objects introduced by inheritance is too static.

Alternatively, the **Role-Object-Pattern** [Bäumer et al., 1997] can be used. In this pattern, the player has a multi-valued association to its roles as separate subtypes of a common supertype. This is a straightforward implementation technique that avoids changing the interface when introducing new extensions. It becomes complicated when dealing with constraints between roles or with shared state. Additionally, the split-object problem needs to be targeted explicitly. Also, clients of the compound object have to deal with role-based dispatch, encapsulation, and object comparison manually [Herrmann, 2010]. Although the concepts of **multiple inheritance** and **traits** can be used to implement roles, they will lead to static systems with an exponential blowup in the number of required contexts [Kühn et al., 2014]. Additionally, parallel object hierarchies occur, where cross-tree constraints are hard to maintain [Fowler, 1997]. **Delegation**, on the other hand, mimics the inheritance mechanism on object level. This requires the generation of management code and can lead to object schizophrenia, too. Finally, **delegation-layers** [Ostermann, 2002] define layers that group behavior for sets of objects and for sets of classes. These layers imply fixed hierarchies and thus, a system design that is too static.

Using pattern-based solutions for implementing roles is subject to a number of trade-offs. Nevertheless, any of the patterns can be the appropriate pattern for a specific problem and with this, a dogmatic adherence to one specific pattern is not the answer. The trade-offs have to be understood and evaluated. The *SCROLL* DSL provides a clean solution for roles at runtime in structured contexts. It does not suffer from the aforementioned problems, such as single, too complex types, object schizophrenia, or the need for hand-written management code.

	Single complex type	Shared state / behavior	Scalability	Interface	Change	Constraints	Object schizophrenia	Exponential blowup	Static design	Parallel hierarchies	Management code
Single Type	■		■	■	■				■		
Separate Type		■	■			■					■
Subtype With Internal Flag	■	■	■	■	■			■	■		■
Subtype With Hidden Delegation	■	■	■	■	■			■	■		■
Subtype With State Object		■	■	■	■			■	■		■
Role Object Pattern		■				■	■				■
Multiple Inheritance / Traits			■		■			■	■	■	
Delegation / Delegation Layers		■			■		■				■

Table 13.1.: Comparison of approaches for establishing dynamic objects at runtime (solely based on [Fowler, 1997]). ■ indicates that there is a problem in the given category. This comparison considers only approaches that do not rely on custom compilers, generators, or other tooling.

13.2. ROLES WITH OTHER PROGRAMMING LANGUAGES

This section discusses how different programming techniques can be used to realize roles. Furthermore, we analyze different role-based languages already in existence and evaluate them with regard to their support for the role features already presented (Table 3.1).

13.2.1. GENERAL PURPOSE- , ASPECT-ORIENTED-, AND SUBJECT-ORIENTED LANGUAGES

Roles may be implemented with **statically-typed, object-oriented languages**. With the *ExpandoObject* [Microsoft, 2016], C# can be considered as the most promising language to provide an alternative implementation of *SCROLL*. The ExpandoObject represents an object that allows for dynamically adding and removing its members at runtime. However, compared to *SCROLL*, this works at another level of granularity. Only single members, like a function or an attribute, can be attached or removed at a single point in time. *SCROLL* allows for grouping those members together (e.g., into classes or case classes) and adding or removing all contained members at once. A better separation of concerns is achieved this way. Manipulating C++'s *vtable* could be an additional alternative to handle the dynamic addition and removal of behavior and structure of objects at runtime. Sadly, vtable mappings from function or structure definitions to implementation locations are always applied to the type of an object, i.e., are valid for all instances of this type. Thus, role-playing could only be achieved on the type level limiting the flexibility of roles.

Aspect-oriented programming allows for implementing cross-cutting concerns via join-points and pointcuts. Often, the composition is done statically although there exist a few dynamic approaches. For instance, OT/J [Herrmann, 2005] uses dynamic aspect weaving at bytecode-level for adding behavior. On the other hand, **subject-oriented programming** can define views on different class hierarchies from different perspectives.

On the downside, it does not support dynamic composition and the set of composition operators is fixed. Furthermore, control flow on the composition itself cannot be specified.

13.2.2. OTHER ROLE-BASED LANGUAGES

Interestingly, most of the existing role-based programming languages are extensions to Java. They are either compiled to Java source code [Østerbye, 2003; He et al., 2006; van der Torre, 2006; Barbosa and Aguiar, 2012] or to bytecode [Herrmann, 2005] directly.

13.2.2.1. CHAMELEON

```
1   class Person {
2     int height;
3     String name;
4
5     void say(String s){
6       System.out.print(name+s);
7     }
8   }
9
10  role Teacher roleifies Person {
11    String getname() { return "Teacher " + name; }
12
13    void teach() { /* ... */ }
14  }
15
16  role Supervisor roleifies Teacher { /* ... */ }
17
18  Person p1 = new Person("Windy");
19  Person p2 = new Person("Bert");
20  Teacher t = new Teacher(p1);
21
22  void accounting(Teacher t) {
23    Supervisor s = t as Supervisor;
24    /* ... */
25  }
```

Listing 13.1.: Code example of Chameleon.

constituent methods

Chameleon [Østerbye, 2003] provides roles through *constituent methods* overwriting methods of their players, which work like advices in aspect-oriented programming. Those are different from normal methods, as they cannot be invoked explicitly. Constituent methods hook on to methods in its player and are executed before, after, or instead of the original methods. They have the possibility to change input arguments and return value. Constituent methods are comparable to advices in AspectJ, whereas pointcuts in Chameleon are defined as hooks for constituent methods. With this technique, an object can be extended without references to roles. When more than one constituent method hooks onto a method, priorities accompanying the constituent methods avoids undeterministic behavior. However, unlike static aspect languages, the priorities are modifiable per object at runtime enabling the order of execution to conform to the objects context. Chameleon has chosen to extend Java due to its wide use. It uses a generative approach. Thus, to transform players and roles, all classes are extended with a role manager field and its instantiation. Then, any direct field access is only viable via accessor calls and accessor methods. Additionally, all method definitions use and remember an explicit self reference. With that, all implicit method calls are changed to call the explicit self reference directly. A role extend its player directly because forwarder

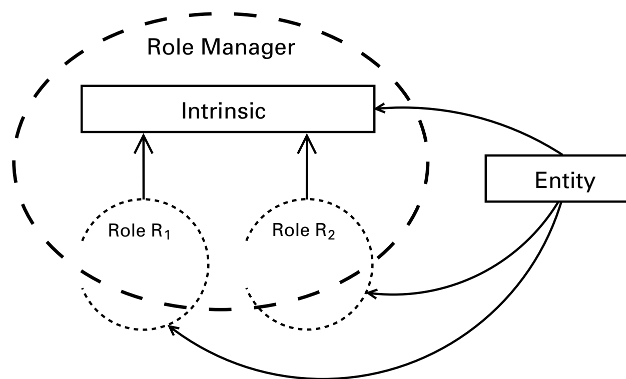


Figure 13.1.: Chameleon handles access to the player by a role manager. This manager intercepts every access. Accessing properties of roles is not intercepted [Østerbye, 2003].

accessor methods are created for all fields which are shared with the player. Finally, forwarder methods for all methods in the player, not redefined in the role, are added.

Listing 13.1 shows an example. There are no changes in the syntax in comparison to Java. The class `Person` is not prepared for roles. The role `Teacher` defines the new property `teach`, and changes how the name field is accessed to include the job title. Field access is converted to method calls as explained above and redefines the method for accessing the `name` field. With that, when invoking `say` on a teacher role, the output will be prefixed. Using the teacher reference `t`, `Windy` may now teach. An `as` operator can be used to see a person through a certain role. As a major drawback of Chameleon, its roles extend the player to gain access to it, which limits the flexibility of roles [Kühn et al., 2014].

13.2.2.2. RAVA

Rava [He et al., 2006] overcomes the inflexibility of roles extending their players to gain access to it by employing the Role-Object-Pattern [Bäumer et al., 1997] extended with the Mediator-Pattern [Gamma et al., 1994].

Rava uses four special keywords to steer the generation of management code to plain Java. `ROLE` followed by a name, defines a new role type. With `roleOf`, the set of classes is configured that roles can be bound to. The keyword `@core` can be used in role methods to access the players methods and attributes. Finally, with the annotation `@INVOKEROLE` parameterized with a role name, or the name of a role method with its arguments, a role method is called. For the translation of roles, the Rava compiler analyzes the tokens by the keywords `ROLE`, `extends`, `implements`, and `roleof`. `ROLE` gets converted into a standard Java `class`. The `roleof` section is removed but every mentioned class is modified to implement `RoleInterface`. In an additional private attribute of a role type, the link to the player is available. A constructor is added to save the role binding rules during the instantiation of role types. Furthermore, a simple access function (`getBindingClasses`) allows for querying binding rules from role definitions. Finally, if a role definition contains the `@core` keyword, it gets converted to the player name bound with the role object.

```
1  class Person {
2
3    String name;
4
5    int deposite;
6
7    public String getName(){
8      return name;
9    }
10 }
11
12 ROLE Employee roleof Person {
13   public int getPaid(int salary) {
14     @core Person().deposit += salary;
15     return deposit;
16   }
17 }
18
19 class Foo {
20   public static void main(String args[]) {
21     Person aPerson = new Person();
22     Integer salary = new Integer(2000);
23     @INVOKEROLE(aPerson, "Person", "Employee", "getPaid", salary);
24   }
25 }
```

Listing 13.2.: Code example of Rava.

Listing 13.2 shows an example [He et al., 2006]. Due to the use of the Role-Object Pattern and generation to plain Java, this solution suffers from object schizophrenia [Herrmann, 2010]. A running compiler was not available online during the development of this thesis.

13.2.2.3. JAVASTAGE

JavaStage [Barbosa and Aguiar, 2012] solves the problem of object schizophrenia by only supporting static roles and is implemented as extension to Java, generating pure Java code by a custom compiler. Roles are directly compiled into the players as inner classes. To avoid name clashes, a customizable method renaming strategy is applied. Its main advantage is the capability to specify a list of required methods instead of a specific player class. This approach limits itself to static roles, unable to represent their relational and context-dependent nature.

Listing 13.3 shows an example written in JavaStage, demonstrating the management of observers in container classes. The subject role of the observer pattern is a container. A generic role is defined for this container and the subject inherits from it. The `FocusSubject` plays the `GenericSubject` role. Methods defined in natural types, i.e., classes, always take precedence over role methods. Conflicts may arise when a class plays roles that have methods with the same signature or when an inherited method has the same signature of a role method. The compiler will issue a warning. Developers can handle these conflicts by renaming the method and calling the intended one. This renaming not mandatory because the compiler uses, by default, the method of the first role in the plays clause order. With this rather static approach, JavaStage does not provide a configurable dispatch at runtime like *SCROLL* does.

```
public role GenericContainer<ThingType> {
  private Vector<ThingType> ins = new Vector<ThingType>();

  public void add#Thing#(ThingType t) { ins.add(t); }

  public void remove#Thing#(ThingType t) { ins.remove(t); }

  protected Vector<ThingType> get#Thing#s() { return ins; }
}

role GenericSubject<ObserverType,EventType> extends
    GenericContainer<ObserverType> {

  requires ObserverType implements void #Fire.update#(EventType e);

  protected void fire#Fire#(EventType e) {
    for(ObserverType o : get#Thing#s())
      o.#Fire.update#(e);
  }
}

public role FocusSubject {

  plays GenericSubject<FocusObserver,FocusEvent>(
    Fire = FigureChanged,
    Fire.update = figureChanged,
    Fire = FigureMoved,
    Fire.update = figureMoved,
    Fire = FigureRemoved,
    Fire.update = figureRemoved,
    Thing = FigureObserver) figureSbj;
}
```

Listing 13.3.: Code example of JavaStage.

13.2.2.4. RUMER

Rumer [Balzer et al., 2007] offers first-class relationships and modular verification over distributed state. Furthermore, it provides several intra-relationship constraints to restrict these relationships. Roles are the named places of a relationship with attributes and methods, but without inheritance. Roles are only accessible within a relationship and not from their player. The existence of an appropriate abstraction to reason about systems composed of classes and relationships is a prerequisite to their specification. As relationships describe the common properties of a collection of groups of collaborating objects, in Rumer they are abstracted as sets of object tuples. Member interposition allows for adding properties to roles (e.g, in Listing 13.4 Line 9). Additionally, Rumer allows defining several inter-, intra and value-based relationship invariants (e.g., `surjective`, `asymmetric` or `irreflexiv`, `forAll`, `isDefined`, or `numberOf`). A running compiler has not been available online during the development of this thesis.

```
1  relationship Attends
2    participants (Student learner, Course lecture) {
3      int mark;
4    }
5
6  relationship Assists
7    participants (Student ta, Course course) {
8      // attribute interposed into role ta
9      String >ta instructionLanguage;
10   }
11
12 relationship WorksFor
13   participants (Student ra, Faculty supervisor) {
14     int >ra grantAmount
15   }
16
17 relationship Substitutes
18   participants (Faculty substitute, Faculty substituted) {
19     invariant surjectiveRelation(Substitutes)
20       && asymmetric(Substitutes)
21       && irreflexive(Substitutes);
22   }
23
24 relationship WorksFor
25   participants (Student ra, Faculty supervisor) {
26     int >ra grantAmount;
27
28     invariant relation(WorksFor)
29       && ra.year > 2
30       && partialFunction(grantAmount) in N;
31   }
32
33 invariant (Attends, Assists) enoughAssistants:
34   forAll c (isDefined(Assists.select(course == c).maxGroupSize)
35   ⟹
36   numberOf(Attends.lecture.select(c)) <=
37     numberOf(Assists.course.select(c)) * Assists.select(course ==
       c).maxGroupSize);
```

Listing 13.4.: Code example of Rumer.

13.2.2.5. SCALAROLES

ScalaRoles [Pradel and Odersky, 2009] is probably the closest relative to *SCROLL*. It is implemented as Scala library as well. With ScalaRoles, a player with temporarily attached role objects is created as a compound object, as shown in Fig. 13.2. The compound object is represented to the outside by a dynamic proxy that delegates calls to the appropriate inner object. Such a dynamic proxy is a particular object provided by the Java API. Its type can be set when creating it through a list of Java interfaces. Internally, dynamic proxies are realized by building and loading an appropriate class file at runtime. The behavior of the proxy is specified reflectively by an invocation handler, that is, an object providing an invoke method that may delegate method calls to other objects. Thus, its type is made up of the core object's type and the types of the role objects that are currently bound to it. The invocation handler of the proxy has a list of inner objects, one player and arbitrary many role objects, and delegates calls to the responsible objects. This is very close to the *SCROLL* approach, but additionally, compiler rewrites are used for calls to role objects. Policies mapping method calls to inner objects may be specified.

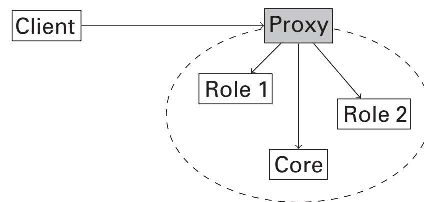


Figure 13.2.: For building compound objects, the dynamic proxy generated will intercept calls and delegates them via an invocation handler [Pradel and Odersky, 2009].

The default is to reflectively delegate to role objects whenever they provide the required method and, otherwise, to the player, such that roles override the behavior of their player. Managing the compound object, creating a dynamic proxy with the appropriate type, and configuring the invocation handler is hidden from the user through a single operator called as. The expression `object as role` allows to access an object playing a certain role by temporarily binding role to object which is semantically close to *SCROLL* 's +-operator. Nevertheless, ScalaRoles lacks an explicit dispatch description mechanism to alter the aforementioned mapping policies for method class. Hence, the developer has to manually write the required management code here.

```
1  trait ThesisSupervision extends Collaboration {
2
3    val student = new Student{}
4
5    val professor = new Professor{}
6
7    trait Student extends Role[Person] {
8      var motivation = 50
9      var wisdom = 0
10     def work = wisdom += motivation/10
11   }
12   trait Supervisor extends Role[Person] {
13     def advise = student.motivation += 5
14     def grade = if (student.wisdom > 80) "good" else "bad"
15   }
16 }
17 // a master student
18 val jim = new Person("Jim")
19 // a PhD student
20 val paul = new Person("Paul")
21 // a professor
22 val peter = new Person("Peter")
23 val master = new ThesisSupervision {}
24 val phd = new ThesisSupervision {}
25 (jim as master.student).work
26 (paul as master.supervisor).advise
27 (paul as phd.student).work
28 (peter as phd.supervisor).grade
29 (peter as phd.supervisor).name
```

Listing 13.5.: Code example of ScalaRoles.

For a role-based application with ScalaRoles, the relation between a student at a university and his supervisor is chosen as example. The student gains motivation when being advised by the supervisor and wisdom when working, and the amount of gained wisdom depends on the student's current motivation. With the class `Person`, supervisor and student can be modeled as roles of it. Listing 13.5 shows a collaboration with two

roles (`Supervisor` and `Student`). A concrete collaboration must extend the abstract trait `Collaboration`. Doing so, it inherits the inner trait `Role` that can be extended by concrete roles.

`Role` takes a type parameter that specifies the type of possible players playing the role. This enables the binding of roles to arbitrary objects, like with *SCROLL*. To use a collaboration, its corresponding trait must be instantiated. Persons are accessed playing a certain role with the aforementioned `as` operator. A role must be qualified with a collaboration instance. The main benefit of instantiating collaborations is that roles may be used multiple times in different contexts. It is also noteworthy that role-playing objects still have the type of their player. Furthermore, a role itself can always access its current player using a method `core`. The state of the roles in a concrete collaboration instance is preserved between different uses of the `as` operator. The practical implementation using Scala's traits as roles reveals the problem that the order of role binding influences the resulting type, e.g., a person playing the father role first and then the student role is another type than the same person playing those roles the other way around. Nevertheless, ScalaRoles can be seen as the predecessor in the spirit of *SCROLL*. The concept of the compound object stems from it, although it is technically implemented differently (dynamic proxies versus `Dynamic` trait with compiler rewrites and implicits).

13.2.2.6. OBJECTTEAMS/JAVA

The most sophisticated and mature approach to roles-based programming so far is OT/J [Herrmann, 2005]. Like Chameleon, OT/J allows overriding methods of a player by aspect weaving on bytecode (both statically and dynamically) and thus, without any source code transformation. It introduces the concept of a *team* representing a compartment whose inner classes automatically become roles (there is no additional keyword for roles). Role definitions can be attached via the *playedBy*-declaration to its players. Notably, OT/J supports both the inheritance of roles and teams whereas the latter leads to family polymorphism [Herrmann et al., 2004]. Every team instance manages its pool of role instance at runtime via caching, role binding is done implicitly once an object enters the corresponding team scope for the first time (implicit lifting). OT/J extends the classical role features (e.g., roles, players, plays-relation) by *callin* and *callout*. The first denotes a method call that is intercepted by the player and forwarded to its roles. The latter describes a method call that is intercepted by a role currently played. This role method then calls the corresponding implementation of its player. OT/J is fully integrated into Eclipse. On the downside, it does neither support multiple unrelated player types for a role type, nor first class relationships, and only a restricted form of constraints. Table 13.2 lists the mapping from the classical role concepts to the implementation approaches used by OT/J.

team

playedBy

callin

callout

In the following, the process of method dispatching within OT/J is discussed. *Lifting* is a translation that considers the dynamic type of a base instance in order to return a role instance of the most suitable role type [Herrmann et al., 2004]. This handles the details of the polymorphic access of role classes. The selection of an appropriate role type commonly precedes the sending of a method to the role. With this technique, roles fit into the general setting of object-oriented methods.

Additionally, some concepts of OT/J are capable for serving for aspect-oriented programming. With regard to the aspect-oriented facilities of OT/J, roles are comparable to aspects. From this point of view, role methods are designated to function like advices. Callin method bindings realize aspect weaving. For higher flexibility and re-usability, it is desirable for role methods to behave like normal methods in object-oriented languages. For an example, consider Listing 13.6 of a tracking aspect for a method `incrXY` for

Reference role concept	Role concept of OT/J
Context definition	Class with team prefix
Context	Instance of a team class
Role definition	Inner class of a team class
Role	Instance of an inner class of a team class
fills-relation	Role definitions bound with the played-by declaration
plays-relation	link between roles and its player
use-relation	Inner class being contained in its outer class
part-of-relation	Mapping from a player to its roles in the cache of a team class instance

Table 13.2.: Reference role concepts and their OT/J counterparts.

FigureElement. The aspect also applies to the subclass Point, where it should behave in a slightly different way as for class FigureElement. For this purpose, the appropriated advice is redefined. OT/J has the ability to face the problem of overriding an advice: callin bound role methods can be overridden in role subclasses which are bound to more special base classes. The role PointTracker inherits from FigureTracker and is bound to the more special base class Point. In the role subclass the method trackIncr is redefined as needed. Calling incrXY() on a FigureElement will now cause an invocation of FigureTracker.trackIncr, while calling the same method on a Point object (or any subclass), will lead to an execution of the overridden trackIncr in PointTracker, because the lifting mechanism lifts each Point to a PointTracker role.

```
1  class FigureElement {
2    public void trackIncr () { /* ... /* }
3    /* ... /*
4  }
5
6  public team class TrackerTeam {
7    class FigureTracker playedBy FigureElement {
8      public void trackIncr () {
9        System.out.println("TrackerTeam : Moving a figure element");
10     }
11
12     abstract int getX();
13     abstract int getY();
14
15     getX → getX;
16     getY → getY;
17
18     trackIncr ← after incrXY;
19   }
20
21   class PointTracker extends FigureTracker playedBy Point {
22     public void trackIncr () {
23       System.out.println("TrackerTeam : Moving the point at (" + getX() + ",
         " + getY() + ")");
24     }
25   }
26 }
```

Listing 13.6.: Code example of ObjectTeams/Java.

Dynamic dispatch can be seen as a mechanism for combining shared and un-shared parts of an implementation. With OT/J, lifting can be seen as an analogy to method dispatch. In the following, a method call on a base instance is considered which is potentially

affected by a callin binding in the system. Before any method is actually invoked, lifting may perform dynamic instance dispatch. This means the stack of active teams that are representing the current context is investigated in order to find the most suitable call target. Method dispatch only needs one type for lookup: the dynamic type of the call target. Lifting, however, requires three pieces of information: a base instance (given at the call site), a team instance (retrieved from the global state of the program as the set of all currently active teams) and a required role type (all role classes that have a callin binding for the particular base method being called). Given that more than one team may be active affecting the same base method by callin, teams are internally kept in a stack, where the most recently activated team has the highest priority for method interception. This priority cannot be configured. Hence, OT/J does not allow for same configurable, dynamic dispatch as *SCROLL* does.

13.2.2.7. POWERJAVA

PowerJava [Arnaudo et al., 2007] is similar to the OT/J approach. It also introduces compartments, denoted *institutions*, whose inner classes represent roles. However, powerJava features the distinction between role interface and role implementation where the former is callable from outside a specific institution and the latter is the institution-specific implementation of the same interface.

institutions

```
1  interface StudentReq {
2    String getName();
3    int getSocialSecNumber();
4  }
5  role Student playedby StudentReq {
6    String getName();
7    void takeExam(int examCode, HomeWork hwk);
8    int getMark(int examCode);
9  }
10
11 interface TeacherReq {
12   String getName();
13   int getSocialSecNumber();
14   int getQualificationNumber();
15   int read(HomeWork hwk);
16 }
17 role Teacher playedby TeacherReq {
18   String getName();
19   int evalHomeWork(HomeWork hwk);
20 }
```

Listing 13.7.: Code example of powerJava (role definitions).

For example (see Listing 13.7), the role `Student` has a `Person` as its player and it is always a student of a `School`. The specification of the capabilities and the requirements of the roles `Student` and `Teacher` are introduced. The roles specify, similar to an interface, the signatures of the methods that correspond to the capabilities that are assigned to the objects playing the role. For example, they return the name of the Student (`getName`), or submit a homework as an examination (`takeExam`). Moreover, a role definition is coupled with the specification of its requirements by the keyword `playedby`. This specification is given by means of the name of a Java interface, e.g., `StudentReq`, imposing the presence of methods `getName` and `getSocialSecNum` (his social security number). Roles add behavior called *powers* in powerJava to objects playing the roles. Power means the capability to modify the state of the institution which defines the role and the state of the other roles defined in the same institution. In the example, the method for taking an

powers

exam in the school must be able to modify the private state of the school. If that exam is successful, the grade should be added to the registry of exams in the school by the teacher. Analogously, the student's method for taking an exam can invoke the teacher's method of evaluating an examination.

Powers seem to violate the standard encapsulation principle, where the private variables are only visible to the class they belong to. However, here, the encapsulation principle is preserved: all roles of an institution depend on the definition of the institution. Therefore, it is the institution itself which gives access to private fields and methods of roles. The role class as inner class is extended with the keyword `realizes` which specifies the name of the role specification, the inner class is implementing.

```
1  class School {
2    private int[][] marks;
3    private Teacher[] teachers;
4    private String schoolName;
5
6    School (String schoolName) { this.schoolName = schoolName; }
7
8    class StudentImpl realizes Student {
9      private int studentID;
10     public int getStudentID() { return studentID; }
11     public void takeExam(int examCode; HomeWork hwk) {
12       marks[studentID][examCode] = teachers[examCode].evalHomeWork(hwk);
13     }
14     public String getName() {
15       return that.getName() + ", student at " + schoolName;
16     }
17   }
18
19   class TeacherImpl realizes Teacher {
20     private int teacherID;
21     public int getTeacherID() { return teacherID; }
22     public int evalHomeWork(HomeWork hwk) {
23       mark = that.read(hwk);
24       return mark;
25     }
26     public String getName() {
27       return that.getName() + ", teacher at " + schoolName;
28     }
29   }
30 }
31
32 class Person implements StudentReq {
33   private String name;
34   private int socialSecNumber;
35   Person(String name, int socialSecNumber) {
36     this.name = name;
37     this.socialSecNumber = socialSecNumber;
38    }
39   String getName() { return name; }
40   int getSocialSecNumber() { return socialSecNumber; }
41 }
```

Listing 13.8.: Code example of powerJava (institution definition).

An institution is a class with inner classes realizing roles in the same way as a class implements an interface. `StudentImpl` realizes the role `Student`, inside the institution `School` (see Listing 13.8). Since the behavior of a role instance depends on its player, in the method implementation, the player instance can be retrieved via the new keyword `that` (e.g., in Listing 13.8 at Line 15). This keyword refers to the object which is playing the

role, and it is used only in the role implementation. To play a role, it is sufficient that the object conforms to the role requirements. Since the role requirements are a Java interface, it is sufficient that the class implements the methods of such an interface. The class `Person` can play the role `Student`, because it implements the interface `StudentReq`.

PowerJava is one of the few research prototypes providing a working compiler. It translates powerJava code to pure Java by means of a pre-compilation phase. Nevertheless, the project has been abandoned [Wielenga, 2013].

13.2.2.8. NEXTEJ

A more recent approach towards context-oriented programming is NextEJ [Kamina and Tamai, 2009] as the successor of EpsilonJ [S. Monpratarnchai, 2008]. It provides contexts as first-class citizens which do not only group roles but also represent an activation scope at runtime. These *context activation scopes* can be nested and act as a barrier where all roles are instantiated and bound automatically.

context activation scopes

```
1  context Building {
2    role Guest {
3      void escape() { /* ... */ }
4    }
5    role Security {
6      void notify() { Guest.escape(); }
7    }
8  }
9  context Shop {
10   role Customer {
11     void buy(Item i) {
12       int p = i.getPrice();
13       Seller.getPaid(p);
14     }
15   }
16   static role Seller {
17     void getPaid(int price) { /* ...*/ }
18   }
19 }
20
21 Building midtown = new Building();
22 Person tanaka = new Person();
23 Person suzuki = new Person();
24 Person sato = new Person();
25
26 bind
27   tanaka with midtown.Guest(),
28   suzuki with midtown.Guest(),
29   sato with midtown.Security() {
30     /* ... */
31     sato.notify();
32 }
```

Listing 13.9.: Code example of NextEJ.

Listing 13.9 presents an example. It features two contexts, `Building` and `Shop`. Inside a building, there are several roles such as `Guest`, `Administrator`, `Security` agent, and `Owner`. Similarly, there are some roles inside a shop, e.g., `Customer` and `Seller`. When a person enters a building, this person assumes the role of a guest. Similarly, the person plays the role of a customer when entering a shop. There are many interactions among roles, e.g., a security agent notifies all guests in the case of emergency, or a seller sells an item to the customer. When leaving a context, the person quits the role played previously.

Furthermore, shops may be inside a building, thus a person may enter multiple contexts at the same time. Inside contexts and roles, methods and fields can be declared, just as with classes. A context can be instantiated with the `new` expression. On the other hand, an instance of role cannot be created explicitly. A role instance is always associated with an instance of its enclosing context. The method call `Guest.escape()` is interpreted as calling the methods `escape()` of all the `Guest` instances. An object entering contexts is created as a class instance. An object enters a context by playing one of its roles. Furthermore, an object can be bound with multiple role instances and can activate or deactivate some of them.

The code beginning from the keyword `bind` (Listing 13.9 at Line 26) is called a *context activation scope*. Before entering the execution scope, it creates role instances and binds them with the corresponding objects, if these objects are not bound with the corresponding roles. If an object is already bound with the corresponding role, this role instance is activated. After entering the execution scope, it is assumed that each object declared in the `bind` clause is bound with the corresponding role instance. For example, `sato` is bound with a role `midtown.Security()`. Inside the following block, `sato` acquires the behavior (and states) declared in `Building.Guest`, thus the method `notify()` declared in `Building.Guest` can be called safely on `sato`. Inside the context activation scope, it is considered that `sato` is a subtype of `Person` and `midtown.Guest`. Outside the context activation scope, it is impossible to access methods declared in roles. This does not mean that the acquired role is discarded outside the scope. Instead, the role instance and its states are retained but deactivated, recovering the original behavior of the object. The retained role instance will be activated again if the object enters the same context again. Additionally, contexts on NextEJ can be nested hierarchically and multiple contexts can be activated at a time. The behavior of a composite context is determined by the order of activation of the constituents. So far, the authors of NextEJ only published their type-system of the core calculus but did not supply a compiler.

13.2.2.9. SUMMARY

In conclusion, it is necessary to investigate how well the implementation of roles at run-time with *SCROLL* blends into contemporary approaches. We use an already published classification scheme from the literature with 26 classifying features of roles [Kühn et al., 2014]. *SCROLL* fully implements 22 of them. A compact overview is given in Table 13.3.

Feature	**Chameleon** [Østerbye, 2003]	**OT/J** [Herrmann, 2005]	**Rava** [He et al., 2006]	**powerJava** [van der Torre, 2006]	**Rumer** [Balzer et al., 2007]	**ScalaRoles** [Pradel and Odersky, 2008]	**NextEJ** [Kamina and Tamai, 2009]	**JavaStage** [Barbosa and Aguiar, 2012]	**SCROLL**
1.	■	■	■	■	■	■	■	■	■
2.	□	⊞	□	⊞	■	□	⊞	□	□
3.	■	■	■	■	■	■	■	■	■
4.	■	■	□	■	■	■	■	□	■
5.	■	■	■	⊞	■	■	■	■	■
6.	□	■	□	■	■	□	□	■	■
7.	■	□	■	■	⊞	■	■	■	■
8.	□	■	□	■	□	■	■	■	■
9.	■	□	□	■	□	■	■	□	■
10.	■	■	■	■	■	■	■	■	■
11.	■	■	■	■	■	■	■	■	■
12.	■	■	■	■	■	■	■	■	■
13.	□	■	■	■	□	■	□	■	■
14.	⊞	⊞	□	□	■	■	⊞	□	■
15.	■	■	■	■	□	■	■	■	■
16.	□	□	□	□	■	□	□	□	□
17.	□	□	□	□	□	□	□	□	□
18.	□	■	□	□	⊞	⊞	⊞	□	■
19.	□	■	□	⊞	⊞	□	■	□	□
20.	□	■	□	■	■	■	■	□	■
21.	□	□	□	■	□	⊞	■	□	■
22.	□	■	□	□	■	□	□	□	■
23.	□	■	□	□	□	□	□	□	■
24.	□	■	□	⊞	■	■	■	□	■
25.	□	■	□	□	□	■	□	□	■
26.	□	■	□	⊞	■	■	■	□	■

Table 13.3.: Comparison of coeval approaches for etablishing roles at runtime based on 26 classifying features extracted from Kühn et al. [2014] presented in Table 3.1. It differentiates between fully (■), partly (⊞), and unsupported (□) features.

14

DISPATCH MODELS

Multiple dynamic dispatch offers several advantages over the classical single dispatch. This dynamic dispatch is not present in current mainstream object-oriented programming languages, such as Smalltalk, C++, C#, and Java. The object model offered by these languages considers methods as operations of a particular class of objects. Thus, an operation always depends on the type of one single object, it is a property of that type and can be encapsulated inside the object. The single dispatch idiom, where functions dispatch on a distinct receiver, consequently models this approach of object-orientation. In contrast, multiple dispatch makes operations depend on multiple different types of objects. This makes it suitable for the use in role-based programming. Other reasons, why multiple dispatch has not been so popular, might be related to the early state of multiple dispatch research at the time, when the above languages were designed. For example, the issue of independent static type checking of separate code modules has been tackled only during the late 1990s. Multiple dispatch is less efficient than single dispatch, due to the complex lookup mechanism, which involves evaluating the types of several arguments, instead of just a single one. Therefore, the additional runtime cost of multiple dispatch is only acceptable, if multi-methods are actually invoked. Its cost should not exceed the cost of hand-coded double dispatch. Finally, the space efficiency of virtual dispatch tables has only been the subject of more recent research, as shown later.

Multiple dispatch was first introduced by **CommonLoops** [Bobrow et al., 1986] and the **CLOS** [Bobrow et al., 1988]. They aimed at extending Lisp with an object-oriented programming interface. The extensions were meant to integrate with the procedure-oriented design of Lisp and facilitate the incremental transition of code from the procedural to the object-oriented programming style. The basic idea is that a CLOS generic function is made up of one or more methods. A CLOS method can have specializers on its formal parameters, describing types (or individual objects) it can accept. At runtime, CLOS will dispatch a generic function call on any or all of its arguments to choose the methods to invoke. The chosen methods generally depend on a complex resolution algorithm to handle any ambiguities, just as in *SCROLL*.

Several more recent programming languages aim at providing multi-methods in more object-oriented settings. **Dylan** [Feinberg et al., 1997], for instance, is based on CLOS. Dylan's dispatch design differs from CLOS in that it features optional static type declarations, which can be used to type generic functions, to constrain their parameters to something more specific than Object, the root of all classes in Dylan. Dylan omits much of the CLOS's configurability. **Cecil** [Chambers, 1992] is a prototype-based programming language that was the first to implement a modular checked static type system for multi-methods. Cecil treats each method as encapsulated within every class upon which it dispatches. In this way, a method is given privileged access to all objects of which it is a part. This is different from CLOS and Dylan where methods are not part of any class. **Diesel** [Chambers, 2006] is a descendant of Cecil and shares many of its multiple dispatch concepts. The main differences to Cecil are Diesel's module system and explicit generic function definitions. As in Dylan and Cecil, message passing is the only way to access an object's state. Diesel uses a modular type system initially designed by Millstein and Chambers [1999] for the Dubious language. The **Nice** programming language [Bonniot et al., 2008] strives to offer an alternative to Java, enhancing it with multi-methods and open classes. In Nice, operations and state can be encapsulated inside modules, as opposed to classes. Message dispatching is based on the first argument

and optionally on any other arguments. **MultiJava** [Clifton et al., 2000] extends Java with multi-methods and open classes. MultiJava retains the concept of a privileged receiver object to associate methods with a single class for encapsulation purposes. However, the runtime selection of a method body is no longer based on the receiver's type alone. Rather, any parameter in addition to the receiver can be specialized by specifying a true subtype of the corresponding static type or a constant value. The MultiJava compiler, translates MultiJava source code into standard Java bytecode. For methods that specialize additional parameters, it introduces cascaded sequences of `instanceof` tests, e.g., for equality comparisons, or value dispatch.

There are many other multiple dispatch languages. **Kea** [Mugridge et al., 1991] was the first statically typed language with multiple dispatch. **Slate** [Salzman and Aldrich, 2005] integrates Self-like prototype-based programming with multiple dispatch to propose a new object model. Some more recent programming languages are designed with multiple dispatch already on-board, among them **Perl 6**, **Clojure**, and **Groovy**. **Scala** supports a form of pattern matching that can be used to dispatch on arbitrary predicates similar to *SCROLL*'s `DispatchDescription`.

Several popular single dispatch languages (e.g., Perl, Python, Ruby, or C++) have been extended to support multiple dispatch, often by means of libraries. **Smalltalk** has been extended with multiple dispatch using its reflective facility [Foote et al., 2005]. **Fickle** [Drossopoulou et al., 2001], a statically typed, class-based object-oriented language with support for object reclassification has been extended with multiple dispatch and first-class relationships [Sinha, 2005]. It augments Java with multiple-dispatch and similar facilities using several approaches. **Parasitic Multi-Methods** [Boyland and Castagna, 1997] is an earlier extension to Java that provides multiple dispatch. Methods defined using the parasitic keyword override less specific methods. A modified compiler translates code that uses the extended semantics into standard bytecode by introducing type testing statements (`instanceof`) to determine the runtime types of all arguments in a method call, thereby dispatching to the most specific parasite.

The **Walkabout** [Palsberg and Jay, 1998] uses the reflection interface of Java (version 1.1 and later) to simplify the implementation of the Visitor pattern. Walkabouts greatly improve extensibility by eliminating the need for visitable classes to implement a `visit()` method and allowing the addition of visitable classes without modification of existing visitors. As the authors note, however, the use of reflection to invoke the appropriate visit method makes this approach impractically slow. The **Runabout** [Grothoff, 2003] improves upon the Walkabout approach in terms of performance. Where the Walkabout uses reflection to invoke visit methods, the Runabout dynamically generates (and caches) bytecode that will invoke the appropriate visitor. This makes the dispatch performance of the Runabout comparable to that of the classic visitor pattern and typically exceeds that of `instanceof` tests.

Dutchyn et al. [2001] modified the Java virtual machine to treat static overloading as dynamic dispatch in classes that implement the provided **MultiDispatchable** marker interface. Millstein et al. [2003] have evolved MultiJava into **Relaxed Multijava**, which essentially allows the programmer to write code in a more flexible style without sacrificing static type checking. While predicate dispatching generalizes multiple dispatch to include field values and pattern matching, *aspect-oriented programming* [Masuhara and Kiczales, 2003] is based around pointcuts that can dispatch on almost any combination of events and properties in a program's execution. Most language implementations summarized here include efficiency evaluations of the respective implementation. Additionally, space and time efficiency of method dispatch has been the subject of a large body of research [Chambers and Chen, 1999; Driesen et al., 1995; Naik and Kumar, 2000;

Publisher	Number of papers
	Raw numbers
ACM	412
IEEE	318
ScienceDirect	156
Springer	518
Others	2698
	After relevance filtering (≥12 citations, only major publishers)
ACM	296
IEEE	149
ScienceDirect	129
Springer	294

Table 14.1.: Overview of the number of papers surveyed with regard to dynamic dispatch published between the years 2000 and 2014.

Kidd, 2001; Zibin and Gil, 2002]. Cunei and Vitek [2005] include a recent comparison of the efficiency of a range of multiple dispatch implementations such as the Visitor pattern, the Runabout, and MultiJava. Studies investigating the practical use of multiple dispatch are less widespread than multiple dispatch implementations. The study presented in Kempf et al. [1987] of the CommonLoops language (a CLOS predecessor) is one notable exception. One of that study's goals was to assess how useful generic functions and multi-methods are for developers. They measured how often these constructs are used in the implementations of CommonLoops itself and a window library called BeatriX.

In the scope of this thesis, we additionally surveyed the research landscape from 2000 to 2014 and took a detailed look at the approaches presented there with regard to dynamic dispatch. An overview of the survey can be found in Table 14.1. Surprisingly, none of the found papers provides dispatch *configurable at runtime*. Nevertheless, in the following sections three selected particular dispatch concepts and modeling approaches are presented in more depth.

14.1. ALIA4J

ALIA4J [Bockisch et al., 2012] offers an approach for implementing advanced-dispatch languages. The goal of ALIA4J is to ease the burden of programming-language implementation resting upon researchers of new abstraction mechanisms. The metamodel of ALIA4J consists of a few well-defined, language-independent abstractions commonly found in advanced dispatching languages (see Fig. 14.1). This acts as a declarative intermediate language for dispatch-related constructs and removes the existing semantic gap between source languages and the intermediate language. During language design and implementation, the metamodel has to be extended with the concrete constructs or sub-constructs used in a language. This allows for reusing the resulting implementation of a construct's execution semantics without constraining the syntax. Thus, the execution environment of the intermediate language can be reused without any constraints on the syntax of the source language.

The novelty of the ALIA4J approach is the declarativeness of the intermediate language, its dedication to advanced dispatching, and its extensibility. To execute code defined in the intermediate language, several back-ends are provided, including platform-independent ones. These instantiate a framework that automatically derives an execu-

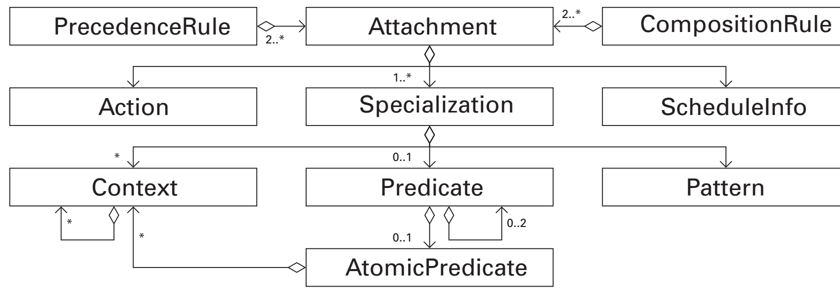


Figure 14.1.: The LIAM metamodel of advanced dispatching from ALIA4J [Bockisch et al., 2012].

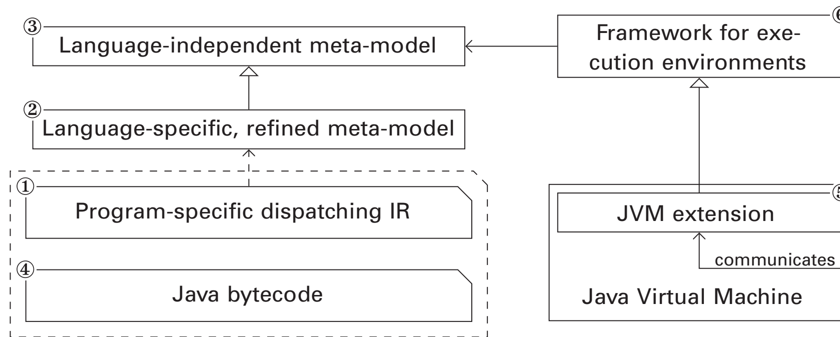


Figure 14.2.: Components and artifacts in an ALIA4J-based language implementation [Bockisch et al., 2012].

tion model from the advanced dispatch's intermediate representation. This, acting as meta-object protocol of advanced dispatching. Several domain-specific programming languages can be mapped to an extension of ALIA4J.

The ALIA4J architecture simplifies the actual implementation of programming languages with advanced dispatching. At its core, it contains a metamodel of advanced dispatching declarations (LIAM) and a framework for execution environments that handle these declarations (FIAL). LIAM defines a language-independent metamodel of concepts relevant for dispatching. Dispatch may depend on predicates based on values in the dynamic context of the dispatch. When mapping the concrete advanced dispatching concepts of an actual programming languages to it, LIAM either has to be refined with the language-specific semantics or existing suitable refinements have to be reused. FIAL defines workflows common to all execution environments able to execute a LIAM-based intermediate representation.

ALIA4J's overall architecture is shown in Fig. 14.2. First, the compiler processes the application's source code and produces an Intermediate Representation (IR) (1) for the advanced dispatching declarations in the program based on the refined subclasses (2) of the LIAM meta-entities (3). Moreover, it also produces Java bytecode (4) for the program parts not using advanced dispatching. Then, at runtime, both LIAM-based IR and bytecode are passed to a concrete JVM extension (5) and are handled by the FIAL framework itself (6).

It should be noted, however, that this approach is purely generative; no configuration or adaptation semantic is available during runtime. Hence, ALIA4J was rejected as underlying base for the implementation of *SCROLL* in the context of this thesis.

14.2. MULTI-METHODS: PROTOTYPES WITH MULTIPLE DISPATCH

In Salzman and Aldrich [2005], the authors describe how a dynamic, prototype-based object model in the style of Self can be reconciled with multiple dispatch. The presented object model, Prototypes with Multiple Dispatch (PMD), combines the benefits of multiple dispatch with a dynamic, prototype-based object model. This leads to a declarative treatment dispatch.

```
object Animal;
object Fish;
object Shark;
object HealthyShark;
object DyingShark;

addDelegation (Fish, Animal);

addDelegation (Shark, Animal);

addDelegation (Shark, HealthyShark);

method swimAway(animal: Animal) { /* ... */ }

method encounter(animal: Fish, other: Animal) { }

method encounter(animal: Fish, other: HealthyShark) { swimAway(); }

method swallow(animal: Shark, other: Fish) { /* ... */ }

method fight(animal: HealthyShark, other: Shark) {
  removeDelegation (animal, HealthyShark);
  addDelegation (animal, DyingShark);
}

method encounter(animal: HealthyShark, other: Fish) { swallow(other); }

method encounter(animal: HealthyShark, other: Shark) { fight(other); }

method encounter(animal: DyingShark, other: Animal) { swimAway(); }
```

Listing 14.1.: The ocean ecosystem example in Prototypes with Multiple Dispatch.

Listing 14.1 illustrates this by example. The behavior of a shark depends on its health. This is modeled as delegation to a `HealthyShark` or a `DyingShark` object. This delegation can be changed, for example, if the shark is injured in a fight. At the same time, behavioral dependence on multiple interacting objects is expressed through multiple method declarations, one for each relevant case. The example does not need additional variables or control-flow branches.

Prototypes with Multiple Dispatch has been implemented in Slate [Salzman and Aldrich, 2005], a dynamically typed programming language. Self [Ungar and Smith, 1987], Cecil [Chambers, 1992], and CLOS [Bobrow et al., 1988] directly inspired the design of Slate and its underlying PMD model.

Two factors were decisive for the rejection of this approach as underlying base for implementing *SCROLL*. First, Slate is not suitable as a host language, as we aimed for a statically typed programming language. Secondly, the internal rank function used in PMD's dispatching algorithm is static and cannot be changed or adapted during runtime.

14.3. KORZ

The Korz computational model [Ungar et al., 2014] provides context-oriented programming by combining implicit arguments and multiple dispatch in a slot-based model. This combination enables writing software that supports both contextual variation with multiple dimensions and evolution to adapt to unexpected dimensions of variability. No additional layers or aspects are needed. Hence, a system programmed with Korz contains methods (*"a sea of methods"*) and data slots. No fixed organization of slots into objects is assumed. Hence, a slot belongs to a number of objects instead of being contained by a single object. Finally, context-dependent objects are formed when multiple slots are joined with regard to their implicit context. At each computation step, a slot is selected from the space using multiple dispatch that is based on the context, a selector, and explicit arguments. At the end that slot is invoked. The context is implicitly passed along during the selection, and hence serves as a set of implicit arguments.

```
1  def {} pointParent = newCoord;
2  def {} point = newCoord extending pointParent;
3
4  var {rcvr ≤ point} x; var {rcvr ≤ point} y; var {rcvr ≤ point} color;
5
6  // defining slot 1:
7  method {  rcvr ≤ pointParent, device  } display { device.drawPixel(x, y, color) };
8
9  // defining slot 2 for adding a dimension
10 method { rcvr ≤ screenParent, isColorblind ≤ true  } drawPixel(x, y, c) {
     {isColorblind: false}.drawPixel(x, y, c.mapToGrayScale) }
```

Listing 14.2.: Example for handling cartesian points with Korz.

Listing 14.2 gives an example inspired by the colored-point example that was quite popular for discussing evolution of object-oriented programs [Ungar et al., 2014]. We start with a cartesian, colored point. First, the program defines coordinates for the prototypical point at Line 1 (from which new points will be cloned) and its parent at Line 2 (methods applicable to all points will be associated). To make the coordinates accessible, the program defines them as the contents of slots. Therefore, two slots are created, each containing a new coordinate. The `point` coordinate is declared to have the `pointParent` coordinate as its parent. The empty slot guard "{}" for a point means that, in any context, an invocation of this slot returns the point's coordinate. With that, the declaration of `display`, consisting of slot guard and method body, defines a method that displays a point (Line 7). The slot guard, {rcvr ≤ pointParent} (Line 7), specifies that the slots are accessible only in contexts in which the `rcvr` dimension is bound to a point, i.e., to a coordinate that is at least as specific as the point coordinate.

Support for colorblind users can be realized as a separate dimension of figures or devices. The programmer only needs to define a more specialized `drawPixel` slot to be used when a new `isColorblind` dimension is present and bound to `true` (Line 10). Whenever `drawPixel` is sent to a context with a screen for the receiver and `isColorblind` bound to `true`, the added slot will be invoked instead of the slot for the `drawPixel` method. This call will map the color to grayscale, and then calls the `drawPixel` method.

Although the dispatching mechanism if Korz is fairly advanced, the slot lookup itself involves attempting to find a single, most specific slot whose guard matches the message. In case multiple targets were found, an error is thrown statically by the system. The developer, by no means, is able to configure that lookup at runtime.

15

CONCLUSION

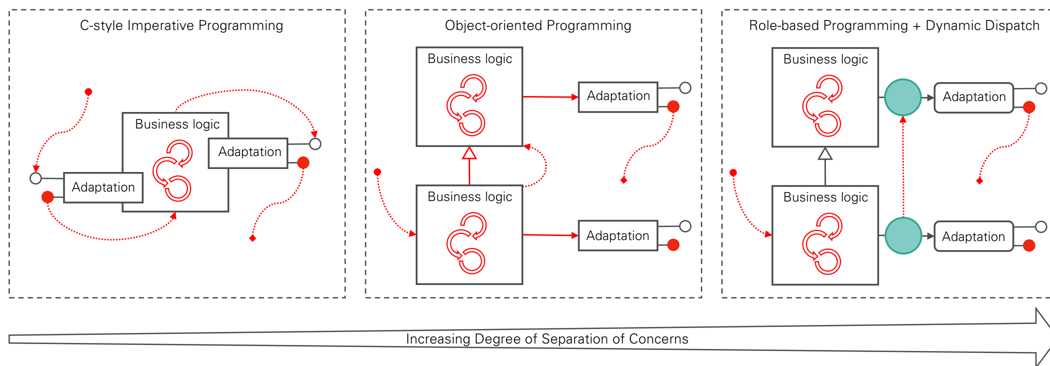


Figure 15.1.: Separation of concerns increases with the introduction of roles and dynamic, role-based dispatch.

In the modern world, software systems are expected to adapt to a changing environment as they become more and more ubiquitous. During their lifetime, new features are requested, existing requirements change, and the hardware and operating systems are regularly being renewed. Software written for a specific purpose may become useful in situations and environments, which the developer did not dare to anticipate. Those situations are ubiquitous in the physical world (e.g., on wearables and smartphones) and ubiquitous in the software world of Internet-based applications. Object-oriented programming, as being widely used to build extensible and flexible software systems, is successful, because it supports programming with data structures that closely resemble the problem domain. However, future software systems expect a higher level of dynamism, not offered by pure object-oriented concepts. With dynamically typed, object-oriented scripting languages, a very flexible programming style is available. Modules, classes and objects can be extended arbitrarily at runtime. But programming in a dynamically typed language comes at a cost. Without static type information, it is impossible to analyze programs statically and catch whole classes of programming errors before actually running the program. The burden is solely carried by the programmer. To cope with challenges imposed by ubiquitous, highly adaptive software systems, researchers proposed several approaches, including the language concept of roles. This concept allows for extracting the context-dependent behavior from the classes and model it in separate role types. Together with role-based, dynamic dispatch, a new level of separation of concerns is revealed (see Fig. 15.1). The core behavior and structure is defined in the object's type. Context-dependent and evolving parts are then specified in role types. Role-playing objects are able to start and stop playing roles to adapt their behavior and structure dynamically during runtime, without the need for reinstantiation. However, role-playing objects need specific forms of multi-dispatch.

Because a declarative and parameterizable approach for four-dimensional, context-aware dispatch at runtime is not yet available, the method-call interception DSL *SCROLL* and its reference implementation *SCROLL* MOP were presented. Firstly, we explain their basic concepts, and in particular introduced the Compartment Role Object Model, as underlying base, in Chapter 3. The foundations for the aforementioned dispatch are discussed in Chapter 4. From a fairly simple example of robotic co-working, we derived several requirements for roles and their proper dispatch at runtime. As it turns out, this dispatch semantics helps to overcome the subtle ambiguities introduced with the rich

semantics of role-playing objects. To introduce all these concepts at runtime, they were implemented with the *SCROLL* MOP, a library for the pure embedding of roles. A pure embedding dos not require additional tooling (e.g., a new or modified compiler, or code generator) as it does not modify or extend the host language syntactically. The size of the library is below 1400 lines of code so that it can be considered to have minimalistic design and to be easy to maintain. Finally, this approach is evaluated, both qualitatively and quantitatively. While fulfilling most of the features offered by the rich semantics of the role concept, the implementation is still too slow, because it heavily uses reflection. To optimize this use of reflection is future work.

The *SCROLL* approach for the pure embedding of roles in a method-call interception DSL, and its dynamic, role-based dispatch is presented in this thesis and consists of the following key contributions:

(1) *SCROLL* and the *SCROLL* MOP This thesis presents *SCROLL*, an embedded method-call interception DSL as library with its underlying MOP [Lämmel, 2002; Mernik et al., 2005; Kiczales et al., 1991] that allows for pure embedding [Hudak, 1998] of roles in a modern, statically typed object-oriented language (Scala) without changing its syntax. It solely utilizes features that are available through the standard compiler. This library allows for easy integration of legacy code and a high separation of concerns. It has a minimalistic design and stays below 1400 lines of code.

(2) A coupling of static and dynamic role typing By relying on a statically-typed host language for roles, *SCROLL* supports the developer with the advantages of static typing and dynamic objects with roles, simultaneously.

(3) A simple implementation pattern for roles in structured contexts The implementation pattern behind *SCROLL* requires three basic components, namely, compiler rewrites, i.e., a compiler-supported variant of method-call interception [Lämmel, 2002], implicit conversions for assembling a compound object from the player plus all of its roles, and a definition table of the relationships between each player and its roles, the role-play graph.

(4) A role-based dispatch configurable at runtime A declarative and parameterizable approach for four-dimensional, role-based dispatch at runtime is presented. This enables the developer to overcome the subtle ambiguities with roles in structured contexts by utilizing an explicit representation of dispatch rules as function objects [Stroustrup, 1995]. The dispatch is based on four dimensions: the name of the computational unit, the context of the receiver, the context of the sender, and, for the first time, on structured contexts. The dispatch can be configured dynamically by node filter functions.

(5) Strong type-safety for role-based dispatch The type checking during role-based dispatch is supported by additional typing information constructed via introspection [Bobrow et al., 1993] and an optional compiler plugin using static program analysis.

(6) The practical applicability Finally, with the application of role-based adaptation for robotic co-working, it is shown how roles as dynamically evolving objects can help to implement highly adaptive systems. With a hybrid automaton, specifying the contexts of the robot, and the four-dimensional dispatch on these contexts, the robot is able to react to unexpected, asynchronous events. The implementation presented with *SCROLL* is simple and demonstrates its basic features and usage.

In conclusion, we have shown how arbitrary objects can be augmented dynamically with new functionality and state grouped together in roles. Moreover, obstacles arising from object schizophrenia can be solved with the concept of a compound object (enabled by dynamic conversions) and an adapted notion of object identity, such that the identity of an object is the same independently of which role is attached. Using Scala's Dynamic trait together with a role-play graph allows for easy querying for behavior that is not natively available to the player. If one is able to find or emulate these three techniques (compiler rewrites, implicit conversions, and a role-play graph) in the desired host language, it is possible to provide an alternative implementation of *SCROLL* in another host language.

16

FUTURE WORK

As every novel approach in the field of programming language design and implementation, *SCROLL* opens a wide space for future work. Several developments are currently work in progress or targeted for investigation in the near future.

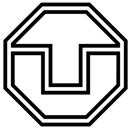
Interdisciplinary collaborations In interdisciplinary collaborations, we aim for other use-cases for the concept of dynamically evolving objects. They should help the domain expert to cope with domain-specific implementation concerns. Specifically in systems biology and, more generally, in scientific computing (e.g., with a Next-Generation Parallel Particle-Mesh Language [Karol et al., 2015]), using *SCROLL* looks promising.

Optimizations With respect to the required performance, optimizations for translating the specific binding and behavior-lookup for dynamic objects need to be developed. A promising direction is the investigation of the `invokedynamic` bytecode keyword introduced with Java 7 to provide an alternative implementation of *SCROLL*.

Other meta-predicates Other dynamic objects, like facets, parts, phases, and aspects could be investigated whether they can be integrated into *SCROLL*.

Dispatching expressiveness With more case studies, it needs to be investigated if the proposed dynamic, role-based dispatch is expressive enough to cope with the requirements of context adaptation. Is a mapping to, e.g., predicate dispatch feasible? What are the benefits, when translating this dispatch semantics into a new role- and context-aware type system? Are existing type systems (e.g., dependent type systems) sufficient?

Dispatch metrics In Muschevici et al. [2008], the authors provide metrics for dispatch (e.g., dispatch ratio, choice ratio, or degree of dispatch). These metrics focus on method definitions and can be measured statically. Tailored to the notion of roles, one could investigate the degree of adaptability provided by the *SCROLL* dispatch concept in comparison to existing approaches.



TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software- and Multimedia-Technology

Software Technology Group

PART V.

APPENDIX

A

A VARIABILITY ANALYSIS FOR ROLES AT RUNTIME

Contemporary literature has not been able to provide a unique definition of what a role is. The various semantics of the role concept have been described in a variability analysis [Graversen, 2006]. This analysis relies on the encountered semantics of roles which goes far beyond the analysis presented in Kühn et al. [2014], because many runtime features are investigated. The following sections analyze role variability with the help of feature diagrams. This approach permits us to reflect over the set of features *SCROLL* covers. The remainder of this section is structured as follows: the first subsection focuses on class instances as role-playing objects because these instances are the fundamental basis of the role semantics of *SCROLL*, while the remaining subsections deeply cover a complete analysis of roles at runtime.

A.1. CLASS INSTANCES AS ROLE-PLAYING OBJECTS

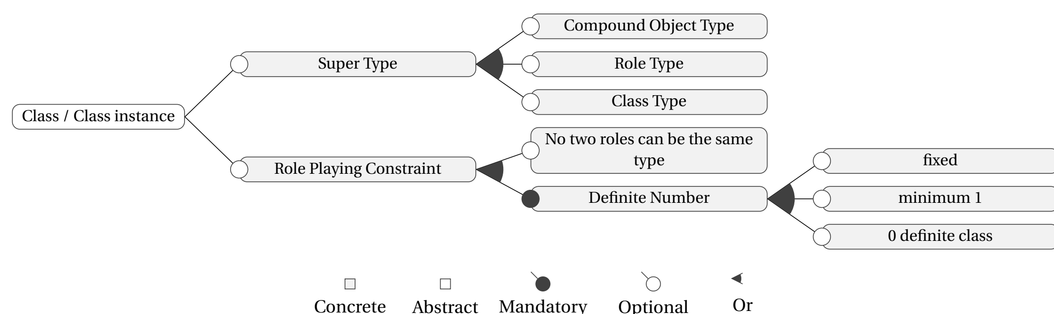


Figure A.1.: Feature model for roles (Class), adapted from Graversen [2006].

Graversen [2006] discusses different entities that can fill (class types, role types, and context types) and play (class instances, role instances, and context instances) roles. The authors refer to programming languages using *prototypes*, where classes are objects. Class instances are able to play roles (Fig. A.1). A class instance playing a role differs a lot from a role instance playing roles. This touches two topics: role-playing constraints and supertypes of those entities, which will be discussed in the following sections.

prototypes

A.1.1. ROLE-PLAYING CONSTRAINTS

Two role-playing capabilities of class instances may be constrained: i) the amount of roles an instance of a class can play may be constrained, and ii) the uniqueness of each role type played may be constrained. With *0 roles*, instances of a class are not allowed to play any roles. Most of the statically typed languages allow classes to be defined as `final`, making them non-subclassable. This offers several interpretations when assuming role types are subtypes of their player:

- Instances of final classes are not allowed to play roles because role-playing is some kind of extension. However, with polymorphic references it is most of the time not possible to statically enforce this constraint. For instance, a reference to a non-final superclass could be used to bind a role.

- A role is not allowed to have an instance of a final class as its player. This can statically be determined, but it is still impossible to detect the situation that an instance of a final subclass of a player is playing the role.
- The concepts of *final classes* (subclassing is not allowed) and *definitive classes* (role-playing is not allowed) are orthogonal concepts. Final constraints static inheritance and definitive constraints dynamic inheritance. Thus, an instance of a final class may play roles.

In most of the role languages (e.g., in [Pernici, 1990; Albano et al., 1993; Li and Wong, 1999; Markovic and Sochor, 2001]) the player cannot exist without at least *one role* being attached to it. In those languages the player only carries an identity. This technique is used to enforce re-usability. With *fixed classes*, the instance of a class is only allowed to play a fixed number of roles. Various languages (e.g., [Smaragdakis, 1999; VanHilst, 1997]) incorporate this feature. Since roles apply to many domains, the only technical reason for this constraint may the potentially more efficient representation of the underlying model. With the last alternative, *N roles*, instances of a class are not subject to any restrictions on the amount of roles they are allowed to play.

Finally, ii) is discussed in the following. This feature is optional and independent of the restrictions under i). The constraint (all roles must be of different types) restricts that a player may only play one role instance of each type of role. Some languages support this feature [Albano et al., 1993; Kniesel, 2000; Truyen et al., 2001].

A.1.2. POSSIBLE SUPERTYPES FOR ROLE-PLAYING

In a role-based programming language, possible supertypes for classes can be class types, role types, or compound object types. That a class is a supertype of a class is common in most role languages. A class is not allowed to extend a role class since a role describes an object in a certain context, while a class only describes objects. Those two concepts are not interchangeable [Osterbye, 1996; Papazoglou and Kraemer, 1997; Steimann, 2000b]. Conceptually, a role cannot be instantiated without a player. So, making a *role class a superclass* to a class requires the class to provide a player for its super. Clearly, this is a misuse of the specialization concept. On the other hand, abstract roles are roles without a player specification. Technically, this brings back inheritance but conceptually, the argument that roles and classes model different things, still applies. In Osterbye [1996], a compound-object type is a subclass of a conglomerate of roles and player. In this purely static concept, the specification of a compound object type must include the roles that comprise the compound object type. BETA's notion of singular static/part object [Madsen et al., 1993] may have been a source of inspiration for compound object types. Additionally, the inheritance chain must be non-cyclic [Kniesel, 2000]. This avoids problematic constructs such as a superclass `S` for class `C` is defined as a direct or indirect subclass of `C`.

A.2. CONSTRAINTS BETWEEN THE PLAYER AND ITS ROLES

This section focuses on the possible constraints between a role and a player, namely, role-playing constraints, player cardinality constraints, role instance cardinality constraints, and dispatch constraints (Fig. A.2). Those restrictions take outset in the "0, 1, ∞" principle [MacLennan, 1983], which argues that a language feature typically is applicable zero, one or infinitely many times. They are orthogonal to each other, and hence can be composed.

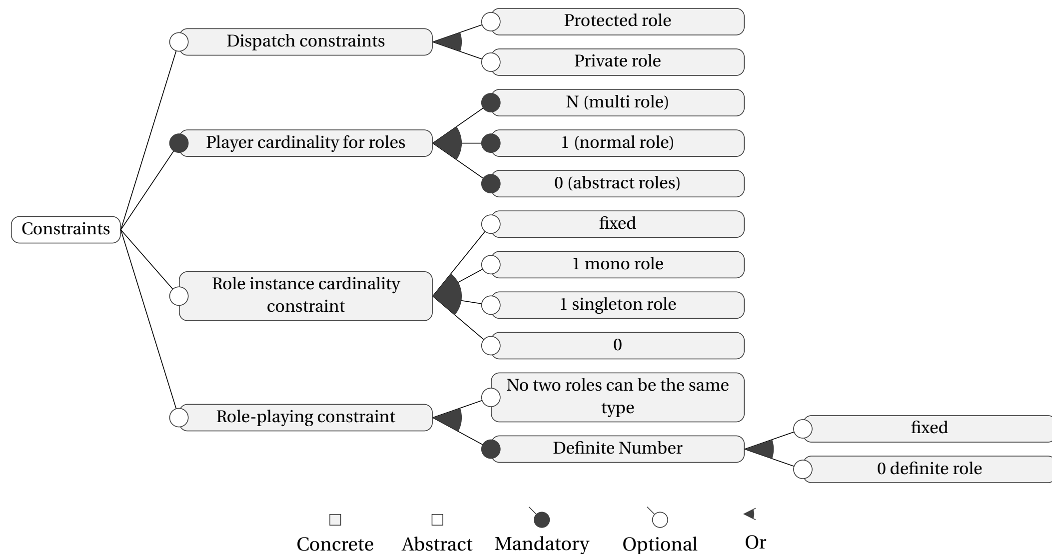


Figure A.2.: Feature model for roles (Constraints), adapted from Graversen [2006].

A.2.1. ROLE-PLAYING CONSTRAINTS FOR ROLES

The following covers the topmost branch of the feature analysis with focus on constraining the cardinality of roles that may be played by a role. With 0 roles, a role is not allowed to play roles. Those roles are called *flat roles*. Alternatively, one could restrict a role to only be allowed to play a fixed maximum number of roles. Various languages [Smaragdakis, 1999; VanHilst, 1997] incorporate this feature. The same argumentation as for fixed classes holds: since roles apply to many domains, the only reason for this constraint may the more efficient implementation. In contrast, with N roles, role instances are not subject to any restrictions on the amount of roles they are allowed to play. Furthermore, it can be prescribed that all roles must be of different types. Some languages support this requirement [Albano et al., 1993; Kniesel, 2000; Truyen et al., 2001]. They rely on role's types for e.g., the method dispatch, acquisition semantics and role creation. Hence, they cannot distinguish between several role instances of the same type. While this restriction is technically possible, it is conceptually not required and various author claims that roles should not be restricted by such a constraints [Gottlob et al., 1996; Østerbye, 2003; Osterbye, 1996; Lee and Bae, 2002; Markovic and Sochor, 2001; Steimann, 2000a; Stein and Zdonik, 1989].

flat roles

A.2.2. INSTANCE CARDINALITY CONSTRAINTS FOR ROLES

The number of simultaneously existing role instances of a given role type may be constrained. This feature is closely linked to the underlying model of the host languages for object creation and destruction. Without garbage collection, e.g., in C++, an often-used idiom is to hand out a copy of the object rather than the object itself. It is questionable if such an instance copy counts for the number of instances set up by the constraint. However, in garbage collecting languages, such as Java or Scala, garbage collection is implementation- and platform-specific. There is an unspecified delay between the moment an object is marked for destruction and its actual removal. Should such an object still be included in the counting? Due to this vague semantics, programmers might want direct support for destroying or moving roles. This results in the following semantic variations. With 0 instances, a role cannot be instantiated. A *singleton role* is a role that can only have one instance at runtime. This may be useful when the Singleton design

singleton role

mono role

pattern would be applied anyway [Gamma et al., 1994]. A *mono role* is a role that only one instance can be played per player or compound object. This may be easily realized by inserting an existence check into the role's constructor, looking for other role instances on the player or the compound object. Another interesting strategy discussed in the literature (e.g., in [Jørgensen and Truyen, 2003; Herrmann, 2005; Harrison and Ossher, 1993]) is to restrict role creation to only occur as a side effect of either casting the player to the role type or using a role cast in conjunction with a player reference. If the player already plays the requested role that role is returned rather than creating a new role. As outlined above, restrictions using fixed numbers represent a technical constraint that is conceptually unmotivated. Contrary to the 0-instances restriction, roles can be instantiated.

A.2.3. PLAYER CARDINALITY CONSTRAINTS

In most of the literature, only the importance of objects playing many roles is discussed. For the sake of symmetry, the analysis of a role having several players is addressed in the following. If a role cannot be played by anyone, it is called an abstract role (*0 players*). This boils down to the concept of classical abstract classes. They are in essence a store for state and code and thus target to be used by inheritance. However, an abstract class is an abstract supertype, but an abstract role is an abstract subtype. In the first possible interpretation, abstract roles do not have a player specification. Technically, this is close to the mixin inheritance paradigm [Bracha and Cook, 1990; Hendler, 1986] and used in some role languages [Smaragdakis and Batory, 2002; VanHilst, 1997]. Steimann's role concept, equating interfaces and roles [Steimann, 2000a,b], is very similar. In the second interpretation, an abstract role is defined as a fully functional role including a player specification, but is marked as abstract and thus, cannot be instantiated. The third interpretation utilizes instantiable roles without players [Lee and Bae, 2002] for the use during testing and independent development. In later stages of the development, an explicit binding is needed to glue them to players. Additionally, methods inside roles may be marked as abstract, requiring an concrete implementation in subroles. The first interpretation is indeed a technical description (i.e., a set of properties and behavior wrapped in a type) intended for reuse. It allows roles to be more generic. The second interpretation allows for a player, hence is written for that specific one. Finally, the third interpretation is motivated by the practical issues during testing and development of roles and players.

multi-role

A role that is allowed to have multiple players is called a *multi-role*. Having several players raises questions. What is a multi-role when very different players are involved? What is the commonality between such entities, and how are they found? What is the practical usage? One motivation for the multi-role is the synergetic effects that may arise when assembling parts. A person and a car can turn into a transport vehicle, a number of cards can turn into a deck of cards usable for games. Such vague descriptions leave room for a variety of interpretations. When using the N player approach as aggregation, it is important to discuss if contemporary aggregation can be represented, like the facade or the mediator design pattern [Gamma et al., 1994]. When a concept from the real-world aggregates several other concepts, it is typically implemented in object-oriented programming languages using delegation rather than aggregation, because the aggregation mechanisms are too simplistic. However, in some collaboration-based languages this approach is used [Aracic et al., 2006; Ovlinger, 2004; Veit and Herrmann, 2003].

An interpretation from the viewpoint of inheritance sees the multi-role as some kind of multiple inheritance. Name collision problems must be handled. Basically, this semantics is closely related to the hubs-Operator in gBeta that assembles several objects connected using delegation [Ernst, 2004]. A multi-role can be seen as some kind of wrapper for

multiple objects, thus it is a kind of facade [Gamma et al., 1994]. The authors in Ernst [2004] define a higher-level interface that makes the subsystem easier to use by providing a unified interface to the interfaces of the subsystem. This may also be implemented using e.g., aggregation or private inheritance. Part hierarchies [Blake and Cook, 1987] extend the interface of the whole by being named entities as facades.

A multiplicity-based interpretation sees a multi-role as a role for a set of objects. Calling a method on such a role leads to an invocation of a set of objects, i.e., its players, like calling an iterator. Ordering (automatic or no ordering) and execution (sequential, quasi-sequential or parallel) can be varied. This semantic is a restricted version of the map function from functional programming.

A collaboration-based interpretation is that a multi-role collaborates with all its players and their roles. This is related to subjects in subject-oriented programming [Harrison and Ossher, 1993]. Previous work [Graversen, 2006] has stumbled upon the legality of going from 0 or 1 to N players for one single role instance, due to the fact that it is hard to define what a multi-role exactly is, as these multiple interpretations shown above reveals.

A.2.4. DISPATCHING CONSTRAINTS

The object may present its roles to the outside itself, or other entities may choose to make the player play a certain role. Hence, it is conceivable to restrict method dispatching in roles. A role may only be viable as target to the base object playing the role, or to the compound object. As a hidden role as base, the following two kinds of role constraints are applicable. In general, roles are public to other entities.

A *private role* on the other hand, is private to its player, enabling it to serve a different purpose. Higher coupling or modeling of internal specialization, such as internal state, may be desired. It separates concerns of the code into smaller units. With the notion of private roles [Graversen and Beyer, 2002] one could temporarily hand out the appropriate role to close friends, a run-time version of the friends mechanism in C++. This would require some kind of security manager. In object-oriented languages, different interpretations of the concept of *private* exist (e.g., Simula: private per object, Java: private per type), hence the definition of private roles may vary. It may be private to the player it is attached to, or private to all instances of the player specification type. In consequence, by using delegation semantics, self is always bound to the player. Hence, a role can never override methods. This leads to the paradox that consultation should be preferred over delegation to allow a role to dispatch on itself. This explicit call from the player to a role is called *role call*. Finally, privacy has another consequence: the player specification for private roles is never polymorphic. The role is not applicable to subtypes of its player because it is only known to the player specification itself.

private role

role call

A *protected role* is public but may not receive any message or state queries from the outside of the compound object it is attached to. It serves as a repository for properties for other roles, very much like an abstract role, but it is not required to inherit from them.

protected role

A.3. RELATIONSHIPS BETWEEN THE PLAYER AND ITS ROLES

The relationships between the player type and the role types it fills are discussed with regard to the inheritance subordination of the role type and the inheritance relationship itself for possible supertypes and subtypes for role types (Fig. A.3).

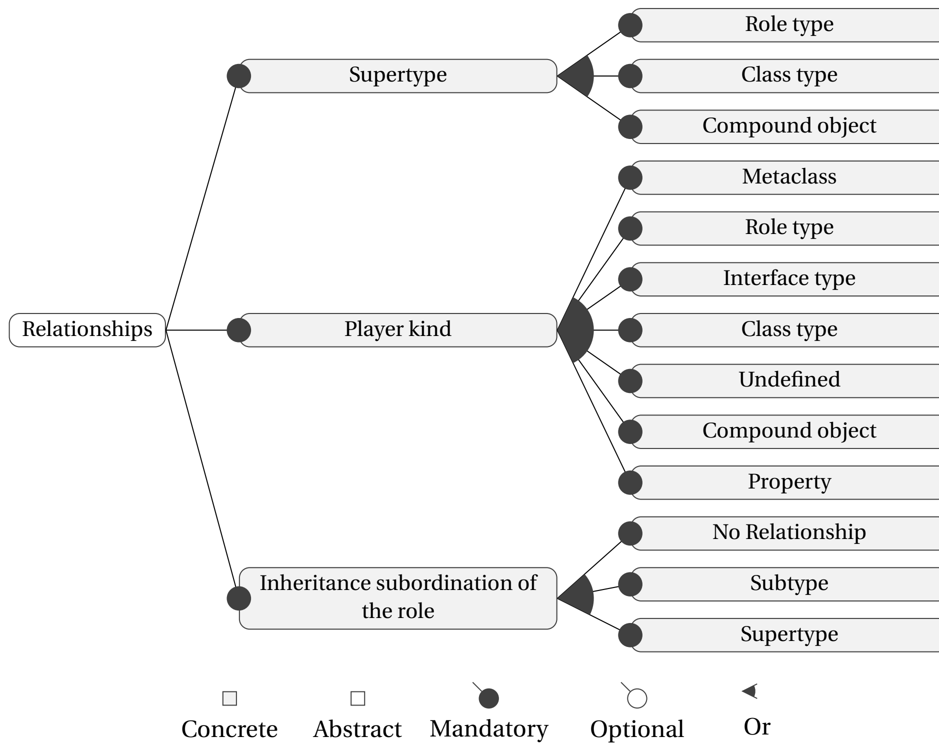


Figure A.3.: Feature model for roles (Relationships), adapted from [Graversen, 2006].

A.3.1. INHERITANCE SUBORDINATION

Often role types are viewed as subtypes of their player types. They may also be discussed as being supertypes or unrelated types to their player types. With roles as supertypes, the statement that a role is more specific than its player is false. For instance, a person can carry many properties that are not required by a customer. *"In fact, it appears that viewing roles as subtypes is a consequence of an inadmissible intermingling of the dynamic nature of the role concept with the static properties of type hierarchies."* [Steimann, 2000a]. Hence, roles are supertypes of their player and more abstract. This may lead to problems with duplicated state since information of the player cannot be utilized by the attached role, as a superclass cannot inspect the state of a subclass. The compiler may collapse those fields by representing them as one. Some C++ implementations [Smaragdakis and Batory, 1998; VanHilst, 1997] utilize mixins. C++'s template mechanism allows the role to access properties only accessible in the player. This results in a lack of separate compilation and thus, a very static role model.

In sum, the following most significant objections to role types as supertypes are as follows. When viewing role types as supertypes, no restriction on the type of its player are made. Thus, it is hard to design a role type for a specific player type. How to express deep roles, i.e., roles for roles? Roles describe an object in a certain context. Hence, a role for a role even further specialized that context. On the other hand, a supertype is more abstract that its player type, and a supertype of a supertype is even more abstract. Those two notions are opposite to each other. If a role type is seen as more abstract, it cannot override its player types methods. And finally, e.g., in C++, virtual methods with the same signature from different role types cannot be resolved by the compiler using typecasting, since only non-virtual methods are accessible this way.

Role types as subtypes is the most common interpretation of the role concept found in the literature. *"The role describes its player in a more specific context than the context*

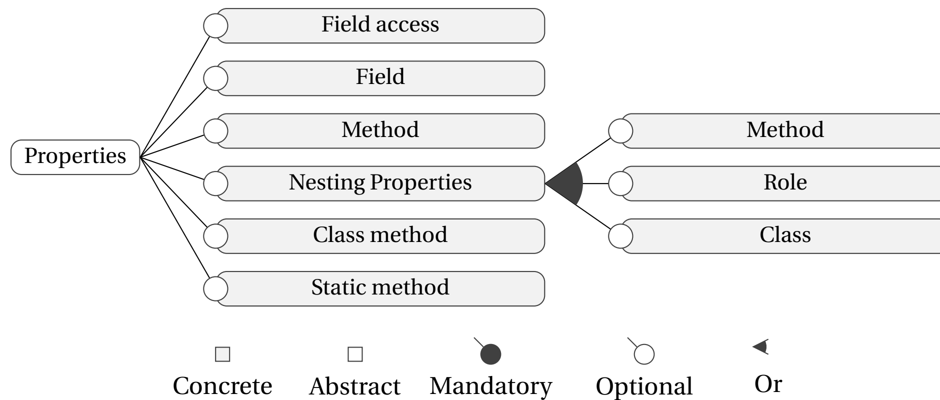


Figure A.4.: Feature model for roles (Properties), adapted from Graversen [2006].

the player is situated in. Thus, the extension is reduced and the intension expanded." [Graversen, 2006]. In dynamically typed languages, roles do not have a player specification [Osterbye, 1996] because these languages rely on the structural layout of objects rather than their type.

Role types as unrelated types to their player types are easier to implement and apply to many unrelated players. This allows for the filtering of the players methods but dynamic dispatch has to be addressed explicitly as, e.g., done with OT/J's *callouts* [Herrmann, 2002]. This enables the role to invoke methods on its player making it versatile and applicable to any object. Another solution is to use method references or delegates. The role constructor takes in addition to the player reference, a set of methods references. Hence, invoking methods on the player without knowing names of those methods is enabled. Using aspects would be a third alternative and is discussed in Sect. 13.2.1.

A.3.2. POSSIBLE SUPERCLASSES FOR ROLE TYPES

We focus now on the investigation of the superclasses of role types. Because *classes* are units without external bindings (e.g., without a player specification), role types may extend them. This allows for code reuse, although classes and role types are conceptually different kinds of entities. This concept is found in the literature [Osterbye, 1996; Gottlob et al., 1996]. Although a compound object exists at runtime, it can be extended like a class in the sense that it does not have any external bindings. This requires a special runtime binding.

A.3.3. POSSIBLE PLAYER TYPES FOR ROLE TYPES

A role type is normally written for a specific player type, but the following alternatives exist. The standard case is indeed the specification of a class type as player. Roles can be attached to subtypes of Java interfaces (as interfaces are uninstantiatable). *Roles* may be roles for roles, which is called *deep roles.* Furthermore, a compound object, created from the players with all of its roles, can be viewed as one object and thus is allowed to play roles. A role can be a role for a property (either a field or a method). Additionally, metaclasses may play roles as well. Finally, a role can be a role for some undefined set of player types.

A.4. PROPERTIES OF CLASSES AS ROLES

State encapsulated in roles involves resource allocation and thus, questions whether roles have an identity. Do they share an identity with their players? What are the boundaries for the notion of roles when allowing inner classes, inner roles, static methods, relationships between roles, and final classes (Fig. A.4)?

A.4.1. FIELDS AND METHODS OF ROLES

When introducing fields and methods to roles, what are their effects and how can those fields and methods be targeted by roles? As soon as we allow for fields in roles, we encounter the problem of name collision like discussed in the literature on inheritance. This problem occurs when an object plays multiple roles containing a field with an identical name. The underlying problem is very similar to diamond inheritance [Taivalsaari, 1996] and can be solved via sharing fields or a complete prevention in the first place.

A.4.2. FIELD ACCESS IN ROLES

With field access, the field access reification problem with a wrong reification strategy for field access in connection with the ability of shadowing fields, and the common ancestor problem arise [Graversen, 2006]. With reification (i.e., the objectification) of field access it is easy for a role to react to changes of its player. This concept is also a part of the aspect-oriented paradigm of programming. The common ancestor problem arises with a form of inheritance, where the same ancestor is inherited multiple times via different inheritance paths. This is also known as fork-join inheritance [Sakkinen, 1989] or diamond inheritance [Taivalsaari, 1996]. The fundamental cause of the common ancestor problem is that inheritance and visibility control are not orthogonal defined in programming languages [Graversen, 2006]. This problem continues in role-based programming. When composing a single type out of a player type and its role types, for each duplicate field and method it must be determined if it is to be shared, repeated, or overridden [VanHilst, 1997]. Also, it may be useful to disallow multiple inheritance completely if fields or methods have not been defined in a common ancestor (to avoid the same name having different meanings) [Albano et al., 1993]. Finally, the general technique of renaming as in Eiffel [Meyer, 1988] can be used to bypass the problem.

A.4.3. STATIC AND CLASS METHODS IN ROLES

static method

A *static method* has no object bound as calling context and hence, no self exists during its execution. Those methods can be called without any instances. Can roles contain such methods and can a role be a role for a player that contains them? All advantages for having static methods, e.g., for sharing values across various instances or having some kind of initialization method, apply for roles modeled as classes as well.

A.4.4. NESTING IN ROLES

nested class
nested method

Nesting of classes (*nested class*) and methods (*nested method*) is supported by many languages, hence it is natural to incorporate this feature into roles as well. It is beneficial over the traditional flat class layout, as it provides another abstraction mechanism. For instance, inner classes implicitly gain an association to their outer instances. Furthermore, exceptions may be encapsulated or helper classes are better hide-able. With roles, it allows for grouping them together into one scope: a collaboration or a compartment.

In languages, like OT/J [Herrmann, 2002] with its teams as first-class collaborations, roles are maintained, allocated and cached within that scope.

Basically, the following three different concepts for classes inside classes are available: nesting, inner, and virtual inner classes. A nested class is a class lexical enclosed by an outer class, while not related to it by any means. Hence, it can be instantiated independently. This semantics can be found in e.g., Python, C++ and C#. An inner class, on the other hand, is a class nested using block structure. They have their enclosing instance available at all times, and thus, cannot be instantiated alone. This is found e.g., in Java and Scala. Finally, a virtual inner class is an inner class but with the addition that an inner class in a subclass may override an inner class of the superclass.

Nested methods (like nested classes) have its outer methods available. The outer method needs to be called first. Within languages with closures (e.g., Python or Scheme) can return a closure from inside a method to hand out an invokable inner method. If methods are implemented as first-class citizens, e.g., in Beta, nested methods can be referenced and passed around. The same holds for Scala, where every method can be interpreted as function object. Most of the time, the nesting of methods is used for source code organization rather that it is founded in the design or conceptual understanding of the problem domain. Hence, roles are not applicable for this concept in the first place [Graversen, 2006].

With inner role, i.e., the nesting of roles inside roles or other classes, the role is hidden or internal to the enclosing context. We differentiate the following interpretations, depending on the kind of entity a role class is nested in:

Role inside roles A nested role is a role for its enclosing roles' player and intended to support the concept of nested classes in the object-oriented part of role languages [Graversen, 2006]. Technical solutions for that (e.g., the implicit attachment of roles by matching the name of inner class, creating instances of inner roles yielding an automatic creation of the inner class, or the creation of an inner role instance during entering the context of the outer role) are in many ways against the principle of the independence of role and player and multiple views of the player.

Roles inside classes In many collaboration-based languages, the class into which roles are nested, offers the collaboration scope, while the roles are placeholders for the actual participants of the collaboration.

A.5. BEHAVIOR OF ROLE-PLAYING OBJECTS

The behavior emerging from the communication between objects in role-based programming is possibly the most challenging part of this thesis. It offers a wide area of semantic variation and needs to be addressed carefully due to its various ambiguities with regard to dispatching method calls from and to roles, calls on self, and calls on other entities (Fig. A.5).

A.5.1. METHOD CALLS FROM AND TO ROLES

A method call essentially is an object sending a message to another object. What those objects are referring to differ a lot in role models in comparison to classic standard object-oriented models. The targets rendered for dispatch perform a lookup of the desired behavior context-dependently. At the end, the gathered methods are executed. Before actually executing a method, we first have to distinguish various semantics of the referencing itself. A *direct references* in terms of role-playing objects is handled the

direct references

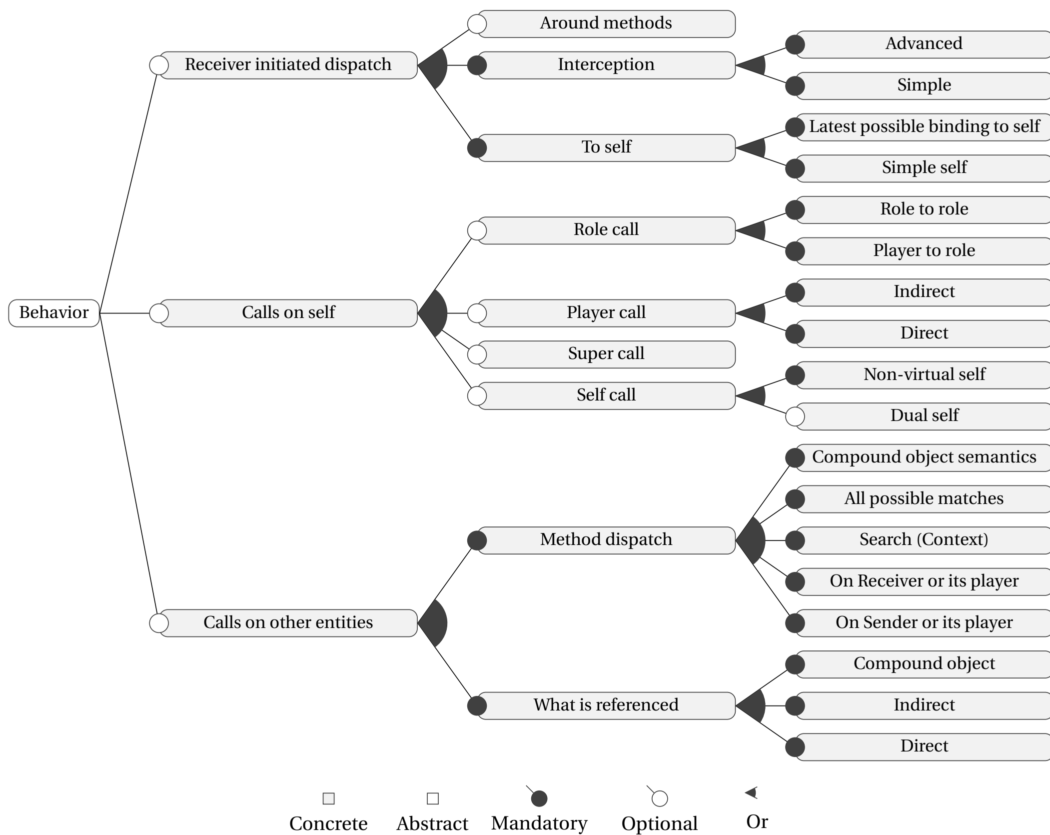


Figure A.5.: Feature model for roles (Behavior), adapted from Graversen [2006].

same way as in traditional object-oriented programming. It is a reference to the object to invoke a method on. In case of role-based programming, it is simply the question of referencing a particular role. An *indirect reference* is usually referring to a facade, i.e., the compound object. The targeted role, or the target method may be hidden in that object. In role-based programming, e.g., when trying to invoke a method on an attached role the reference may instead always point to the player. A compound object is a player with a set of role instances attached. Hence, a reference to that always references multiple objects at once. This is called *compound object reference*. This reference is static, but the referred compound object may not. For instance, Osterbye [1996] with their Smalltalk implementation allow for adding instances to this compound object, but not for removing them. This enables viewing the same object from different perspectives at the same time. Internally, compound objects contain many objects and delegation is used for dispatch. To our knowledge, Bardou and Dony [1996] were the first thinking over a sophisticated method lookup mechanisms for those split-objects yielding the correct reference when trying to find the desired behavior. In role-based programming we are facing the same problem. When building the compound object out of the player and all of its roles, references may not point to the innermost player or some particular role directly.

indirect reference

compound object reference

What happens when a method is called on a reference, i.e., a message is sent to some receiver? Referring to an object and dispatching a message according to the underlying implementation lookup table, is the traditional way in object-oriented languages (e.g., via a v-table, like in C++) and is called *sender-side specific dispatch*. In role-based programming, in addition to the super hierarchy, the set of bound roles is considered. This can happen at compile- (e.g., [He et al., 2006; van der Torre, 2006; Kamina and Tamai, 2009;

sender-side specific dispatch

Barbosa and Aguiar, 2012; Li and Wong, 1999; Albano et al., 1993; Herrmann, 2002]) or at runtime (e.g., [Pradel and Odersky, 2009]). Dispatching depends on either the static or the dynamic type of the reference [Büchi and Weck, 2000]. When using the class type for a reference, code from that class should be executed. For role types, the same applies for executing code from that role implementation [Jørgensen and Truyen, 2003]. The major problem with that static approach is that this qualification relies on the type information, rather than the instance information based on the player with all of its roles attached.

lookup strategies

Different *lookup strategies* do exist in the literature. With predicate dispatch [Chambers, 1993; Ernst et al., 1998], methods can be overloaded by guard predicates that are checked at runtime and determine if the method is allowed to be executed. Ideas for combining roles with predicate dispatch can be found in Odberg [1994]; Papazoglou and Kraemer [1997]. The concept of predicate dispatch requires static completeness checks, a mechanism to avoid side-effects during evaluation, and the reduction of the combinational complexity. As an alternative, double lookup [Albano et al., 1993] can be applied. In that concept, a method lookup on the compound object is performed, investigating all roles, descendant roles (deep roles), and the player itself. Problems like self-recursion and ambiguities are resolved at runtime. This leads to a very flexible understanding of role dispatch.

acquisition dispatching

Sending a message to the latest attached role is called *acquisition dispatching* [Graversen, 2006]. This approach supports state-dependent behavior. Instead of using branching (e.g., with `if`-constructs, which is too monolithic, static, and makes code difficult to modify and extend [Gamma et al., 1994]), the corresponding code is separated into roles. After that, state change is modeled using role-playing. As an important downside to that approach, objects cannot play multiple roles from different contexts and in case of signature or naming clashes always the last attached role will be invoked. This massively limits the expressiveness of roles.

Another approach is the sequential re-sending of the message to all roles an object plays [Van Paesschen et al., 2004]. This broadcasting reaches all of the players role behavior, which matches the required signature. This approach simply ignores naming and signature clashing, hence, leads to their automatic resolution. However, it is difficult trying to implement an object in various contexts since one as to manually at runtime activate and deactivate the visibility of roles and store that knowledge (e.g., in dynamically growing and shrinking lists). This additional overhead may not be desirable.

When referring to a compound object (implying *compound object semantics*), one potentially refers to many role instances on the same player. Calling a method on that compound object may yield different results depending on the selection. In sum, sending a method to a compound object will result in a dispatch to all roles attached in addition to the innermost player specification. To prevent ambiguities due to name and signature clashes, attaching roles with the same methods or attributes is disallowed in all role language implementations up to our knowledge. For instance, the language definition for OT/J states: *"A role class may not have the same name as a method or field of its enclosing team. A role class may not shadow another class that is visible in the scope of the enclosing team. [...] Along implicit inheritance, the names of methods or fields may not hide, shadow or obscure any previously visible name."* [Herrmann, 2016]. As another solution, renaming strategies can be applied. As a consequence, reasoning about the program becomes a lot harder and entails subtle problems with regard to typing and type inference.

A.5.2. SELF CALLS IN ROLE-PLAYING OBJECTS

Typically, self is the initial receiver object of a compound object. Using roles as conjunctive attachments, one of those roles is the initial receiver. This enables changing state

whilst executing one of the objects' methods [Graversen and Beyer, 2002; Kniesel, 2000]. During role-playing, *dual self* may refer to two different objects: the initial receiver (the whole compound object) and the currently executing part object (a role instance). On the other hand, the *non-virtual self* strategy interprets player and their roles as tuples and uses no compound object semantics. This concept is used in various role languages (e.g., [Herrmann, 2002; Aracic et al., 2006]) and yields a very flexible understanding of roles.

A.5.3. SUPER CALLS IN ROLE-PLAYING OBJECTS

As reuse concept, in statically typed programming languages, the super call is not late bound. Mostly, it is implemented by replacing it with a non-virtual call at compile-time. While in static inheritance, a subclass instance is represented by one object, in role-based programming languages there are at least two (the role instance and its player instance). This makes implementing a role-aware super call impossible, since it requires to perform a static method lookup which is not context-aware at all. Additionally, the class's immediate super does not necessarily hold the method called and it must be ensured that subsequent super calls start are searching for the target method in the right context.

A.5.4. PLAYER CALLS IN ROLE-PLAYING OBJECTS

direct player reference

Calls to the player are made on its references, either directly, or as an indirect reference [Graversen, 2006]. With the non-virtual self strategy, the player reference is similar to a reverse aggregation. With a *direct player reference* one always directly refers to the player instance. This is used on almost all role-based programming languages (e.g., in Fowler [1997]; Gottlob et al. [1996]; Østerbye [2003]; Hirschfeld [2002]; Osterbye [1996]; Lee and Bae [2002]; Markovic and Sochor [2001]; Richardson and Schwarz [1991]; Schrefl and Thalhammer [2004]; Herrmann [2002]; Kamina and Tamai [2010]).

A.5.5. ROLE CALLS IN ROLE-PLAYING OBJECTS

Role calls allow for the invocation from the player to one of its roles, or roles among each other within the compound object. Usually, in the first alternative, the player is not allowed to do so because it does not, and should not, know about its roles. Otherwise, it would depend on its roles which violate the whole abstraction concept of role-based programming. Nevertheless, gaining knowledge about the set of roles currently bound to a specific player is unavoidable in certain situations (e.g., interacting or cooperating roles) and beneficial for testing dependencies between roles, or setting up restrictions.

A.5.6. AROUND-METHODS IN ROLE-PLAYING OBJECTS

The notion of an around-method is a well-known concept [Moon, 1986; Kleene, 1989]. In the context of aspect-oriented programming, around-methods are used to separate and modularize code crossing inheritance hierarchies. Around-methods can easily be enabled and disabled during compile- and runtime. Roles achieve the same kind of separation and support instance level composition of aspects [Graversen, 2006].

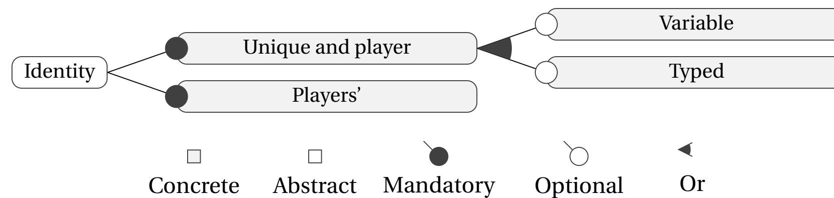


Figure A.6.: Feature model for roles (Identity), adapted from Graversen [2006].

A.6. IDENTITY OF ROLE-PLAYING OBJECTS

The importance and meaning of object identity has been discussed for a long time, in particular in the 1980s and 1990s. The purpose of identity is to offer a representation of the individuality of an object independently of how it is accessed [Khoshafian and Copeland, 1986]. Objects localize state and behavior, and the identity is the single key to access them [Booch, 1991]. Hence, the identity is the most important property of an object [Sakkinen, 1989]. This identity must be unique and permanent within the given universe of objects. It is differently supported in the language constructs inheritance and aggregation. For static inheritance, the class and its superclass share one identity per instance. On the other hand, aggregation builds a whole from a certain number of parts. They maintain their individual identities. This difference is important when defining identity for roles (Fig. A.6).

A.6.1. ROLES WITHOUT UNIQUE IDENTITIES

Older role papers and languages view roles as supertypes of their players, hence they share one identity (e.g., in [Osterbye, 1996; Stein and Zdonik, 1989; Albano et al., 1993; Richardson and Schwarz, 1991; Lee and Bae, 2002; Gottlob et al., 1996; Bardou and Dony, 1996; Papazoglou and Kraemer, 1997]). But also newer models use this strategy (e.g., in [Steimann, 2000a; Chernuchin and Dittrich, 2005]). This approach has strong relations with static inheritance as described above. In these works, the consensus is that roles do not to have a unique identity, because they are not considered to be an independent entity. The player and its roles share one identity and are seen and manipulated as a whole. Some authors argue [Papazoglou and Kraemer, 1997; Dahchour et al., 2002] that a role-playing object should retain its identity, otherwise serious problems arise when other objects in the system try to contain references to the role-playing object. In [Bardou and Dony, 1996] roles are never directly accessible. They cannot be passed along in a method call and dispatching to roles occurs through the innermost player. Those approaches make sure that when a role passes itself along in a method call, that reference will always point to the innermost player. This cannot be done simply be redefining the meaning of that self reference when role methods are executed since this will break self calls within the role. Alternatively, a special reflective method to perform self calls can be provided. Many tasks become counter-intuitive with the lack of a unique identity such as comparison, caching, logging, distribution or other low-level operations.

A.6.2. PROBLEMS MOTIVATING A SHARED IDENTITY

The most stated problems with roles having a unique identity are the part/whole problem, object schizophrenia and that comparison of identity breaks encapsulation. Object schizophrenia [Harrison, 2016] with compound objects arise when state and behavior of what is intended to appear as a single object are broken into several entities and where

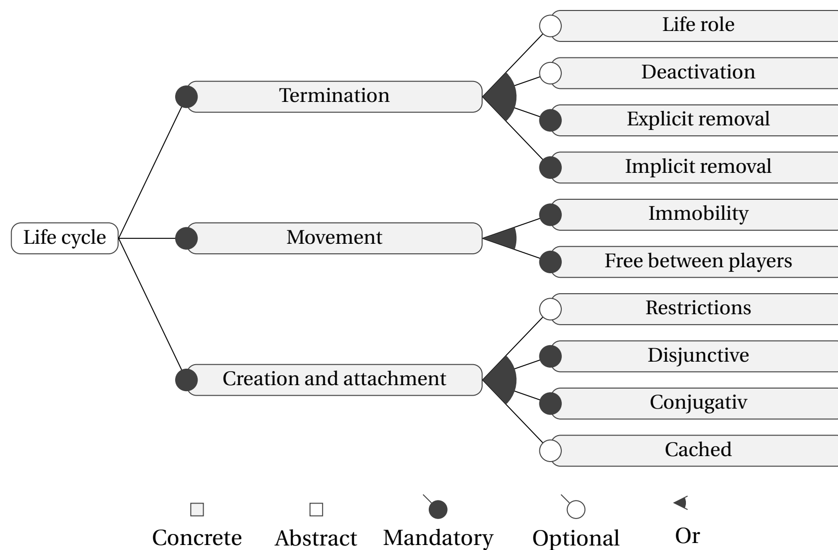


Figure A.7.: Feature model for roles (Lifecycle), adapted from Graversen [2006].

each of these has its own identity. It can be seen as another instance of the split-object problem. Causes of object schizophrenia are broken delegation (when consultation is used but delegation was needed), the player has another identity than its roles, the player may have its methods called and state changed without the knowledge of the roles, and when a *doppelgänger* (several part objects of a compound object subscribe to the same service as different objects) appears.

doppelgänger

A.6.3. ROLES HAVE IDENTITIES

When roles have an identity of their own, the problems mentioned above may arise. With the notion of identity separated from the notion of references, i.e., an identity check is no longer a reference comparison, the problems no longer arises. Identity lookup realized as identity comparison uses a method call to an identity method. Furthermore, object identities may be context-dependent [Sousa et al., 1995]. Two object may have the same identity for clients while for the system (e.g., the one performing distribution, caching, etc.) they have different and unique identities.

A.7. LIFE CYCLE OF ROLE-PLAYING OBJECTS

The life cycle of roles is discussed in the following with regard to role creation and attachment, role movement, and role termination (Fig. A.7).

A.7.1. ROLE CREATION STRATEGY

Roles can be taken on by the player or be attached from the outside [Graversen, 2006]. The implications are influenced by the following two questions: how is the role attached (conjunctive or disjunctive) and what kind of role is used? In addition to that, the creation of a role might not allocate new memory. With cached role creation or moving an old role onto the player, semantics of *object pooling* are achieved. In some role-based programming languages [Odberg, 1994; Kniesel, 2000; Jørgensen and Truyen, 2003; Aracic et al., 2006; Herrmann, 2002], the creation of roles is achieved implicitly (e.g., using a cast). Thus, a concrete role instance is returned rather than being newly created if the

object pooling

player plays it already. Roles that are no longer referenced, may be cached rather than being garbage collected. For the subsequent role creations and attachments, the role cache will be inspected first for reusing existing role instances. Furthermore, the creation of roles may be superimposed (i.e., automated). This is available under the term *lifting*, e.g., in Herrmann [2002]; Aracic et al. [2006]. To only allow for role creation under certain conditions, it may be restricted by additional predicates [Herrmann, 2005].

lifting

A.7.2. ATTACHMENT STRATEGIES FOR ROLES

Together with creating a role it has to be played by some player. As an exception, some role languages (e.g., Lee and Bae [2002]) allow to have role in existence without a player. With *conjunctive attachment*, it does not matter if the new role is attached directly to the innermost player, or to any of the previously added ones. This offers the following advantages. It solves the *spaghetti reference problem* [Jørgensen and Truyen, 2003], where additional roles to already used players are added by other clients. These players and roles need to be updated and book-kept very carefully to ensure consistency. And finally, it solves the compound object interception problem, because it is ensured that at least the methods in the latest bound role will intercept the arriving messages. On the downside, the following problems arise with this strategy. It lacks multiple, separated views, and the diamond inheritance problem shows up. One cannot implement a player playing two roles of the same type. Finally, roles cannot play roles because they are attached to the compound object and not to specific roles. This implies that all methods in roles must invoke their player, since some other role that was attached earlier may rely on being called.

spaghetti reference problem

The *disjunctive attachment* of roles is the opposite strategy making the role destination explicit and enabling multiple views on the subjects. Roles can be attached to the innermost player or to any of its roles allowing it to play several roles of the same type. This attachment to different parts of the compound object leads to some problem with regard to identity: *"this approach is completely not feasible in object-oriented programming. This is because one cannot rely on the identity of the wrapped objects [the compound object] anymore. [...] When an already operational object needs to be wrapped with a new combination of wrappers; here it is not known in the general case which other client objects have references to this object and which of these objects need to have access to the newly created wrapper chain"* [Jørgensen and Truyen, 2003]. This problem might be solved with a combination of: i) a notion of identity, and ii) when asking the compound object or a part object to return a role of a specific type, the request is sent to the compound object, utilizing a full transitive traversal of all roles. This solution is suggested by various authors [Gottlob et al., 1996; Schrefl and Thalhammer, 2004]. In case of the ambiguity having several roles of the same type attached, simply a list is returned.

disjunctive attachment

A.7.3. ROLE MOVEMENT

The following section describes the ability of roles to move between different players and contains technical details in various situations. Moving roles enables adaptation and reflects the dynamics of real-world applications. It is seen as dynamic re-classification [Graversen, 2006]. For clients of players that have their roles moved, subtle problems with regard to the history arise, as role movement should run in a transaction, i.e., is a history-free operation. Hence, references to roles may become invalid. In most situations where the client only cares about the behavior a role offers, rather than the specific player, moving a role is not problematic. In the following, some problematic, dynamic situations for role movement are discussed. When moving roles that do not play roles, one has to

think about two situations. First, moving a role not executing any method, and secondly, moving a role currently executing one of its methods (i.e., still has one of its methods on the call stack). In the latter, ensuring error free execution is not possible. It might be executing its behavior in the wrong context. To avoid inconsistencies or *partly frozen object state* [Kniesel, 2000] the developer has to ensure the termination of a role's method manually. Moving a role that plays roles along all other roles played by the player (i.e., deep roles) is a matter of consistency and compliance to constraints. While obtaining consistency is usually no problem (the number of roles played in sum is the same before and after the change), staying compliant to constraints remains difficult. To solve the updating problem and avoid inconsistencies, the set of players, a role targeted for movement has, should be notified via a subscriptable event. The role-based programming language should provide appropriate hooks for that purpose. To our knowledge, none of the contemporary literature on roles addresses this.

partly frozen object state

Obviously, some combinations of role movement are not allowed from a conceptual point of view and are target for transition constraints. For example, it should be impossible to move the PhD role to a person that is no student. Several constraints mentioned in the literature are noteworthy. One could specify a history of adding and removing roles [Su, 1991], use pre- end post-conditions [Li and Wong, 1999], use transitions as first-class constructs [Pernici, 1990], or use categories and predicates [Odberg, 1994; Chambers, 1993]. Finally, in Riehle [2000], four role constraint are presented: role-imply, role-equivalent, role-prohibited, and role-dontcare. This constraints allow for constraining the relationships between roles and are beneficial in the context of role movement. Not allowing moving roles at all would increase predictability of the implementation [Büchi and Weck, 2000]. Static role models implementing roles as C++ templates and compiling them into the player are examples of such an approach (e.g., in Smaragdakis and Batory [2002]; VanHilst [1997]). This leads to rather static role models.

Roles may be terminated implicitly by a garbage collector, or explicitly by the program. With roles being attached and removed from its player both solutions have a conceptual and technical problem with the absence of history with regard to role transfer and removal. For instance, some clients may have problems if the temporary extension of a person as a student disappears suddenly. Those transient information will be lost. Hence, role replacement could be an option [Gottlob et al., 1996; Albano et al., 1993]. An *explicit removal* of roles leads to their termination which is only possible if that role has none of its methods on the call stack. With deep roles, this might be difficult to determine. A less restricting solution is to activate and deactivate roles rather than deleting them [Graversen and Beyer, 2002; Papazoglou and Kraemer, 1997; Herrmann et al., 2004; Richardson and Schwarz, 1991; Jørgensen and Truyen, 2003]. An *implicit removal*, when no one refers to a role instance anymore, is done by automatic garbage collection through the JVM. Hence, memory is freed and dangling pointers, memory leaks or call stack issues with currently executed methods are no problem at all. The semantics with regard to a role object is the same as with any other object within the rest of the host language. On the other hand that might be conceptually (e.g., is a person really a student if no institution has the person enrolled?) and technically (there is no guaranteed garbage collection time and explicitly steering it is extremely costly and no static assumption can be made on the JVM) problematic and is open for further research. In a unified approach, in which the implicit and explicit solution for role removal is both incorporated and can be used by the developer simultaneously, the code might be difficult to read and understand. In some situations explicit role removal needs to be applied, whilst in other situations the removal is handled automatically.

explicit removal

implicit removal

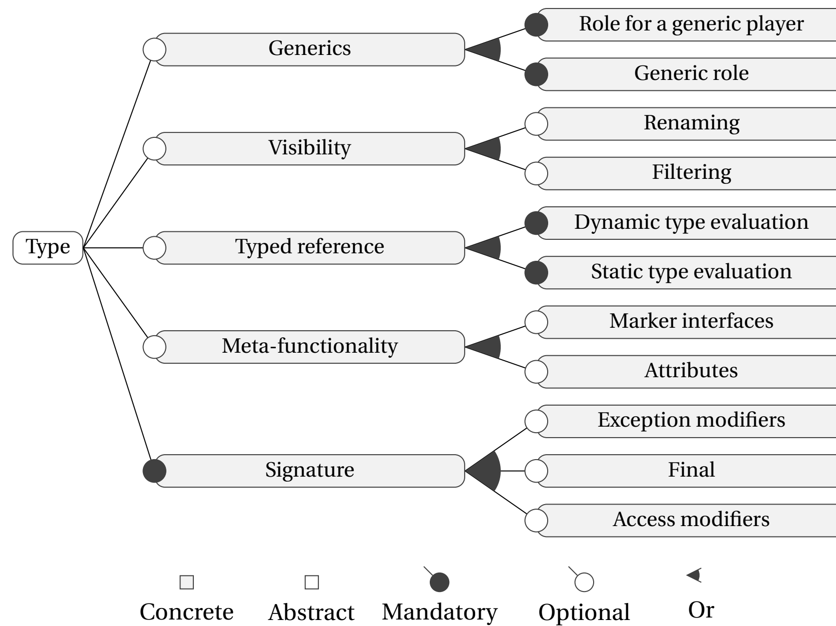


Figure A.8.: Feature model for roles (Type), adapted from Graversen [2006].

A.8. THE TYPE OF A ROLE

Roles in dynamically and statically typed languages differ [Graversen, 2006] (Fig. A.8). Both types of host languages provide basic concepts like the notions of signatures, exceptions, access modifiers and attributes. In statically typed languages, a re-definition of the existing type system is needed to ensure type-safety when constructing role-playing objects, e.g., to disambiguate method dispatch and ensuring proper attachment of roles. In the following, the aforementioned concepts are explored and discussed in more detail. Furthermore, to tackle practical problems when actually programming with roles, we discuss the parts of the signature of a method. Although, statically typed languages typically define a large set of access modifiers, we restrict this part of the analysis to public, protected and private as they are most commonly used. Which properties in the player are actually visible to the role and can those properties be overridden?

final class

A *final class* states that it cannot be subclassed but this may not affect dynamic subclassing with roles. With delegation, a role can re-define an otherwise final method of its player. While this is conceptually irrational due to consistency or security concerns, it favors pragmatic issues over type-safety. In Lasange/J [Jørgensen and Truyen, 2003], instances of classes containing final methods can play roles overriding them. OT/J [Herrmann, 2005] ignores the final annotation on classes and methods by default. In some languages, *checked exceptions* are implemented. They are part of the method signature. This list of exceptions is covariant with regard to inheritance. Thus, methods overridden by a role should only be allowed to define more specific exceptions but not introduce new ones. Some role languages support exceptions, e.g., Büchi and Weck [2000]; Kniesel [2000]; Truyen [2004], but all of them disallow roles to throw exceptions. With *meta-functionality*, e.g., Java's `Runnable`, `Clonable` or `Serializable` marker interfaces, the functionality of the class these interfaces are attached to, is altered. This leads to object schizophrenia in composition with roles because these marker interfaces may affect the player, the compound object, or other currently bound roles. Roles are used to represent a part of the player, but when applying such a marker interface, they may need to represent the whole. The same holds for marked attributes, but on a more fine-grained level. Using

meta-functionality

roles as filters allows for filtering incoming, as well as on outgoing messages during the communication with and within role-playing objects. Hence, these filters can be applied on the sender and receiver side. Eiffel contains strong *renaming* capabilities [Meyer, 1988] to avoid signature clashing. Within role literature [Richardson and Schwarz, 1991; Herrmann, 2005], renaming is mainly used to preserve access to information of the player which otherwise would be overridden by the role. Finally, *parameterization* makes roles more type-safe and more reusable.

B

AN OVERVIEW OF SCALA

This chapter contains a short introduction to Scala [EPFL, 2016c], both a pure-bred object-oriented and fully-blown functional language build on-top of the JVM. It is statically typed like Java, but allows for automatic type-inference in most cases. It smoothly integrates with existing JVM-based code and tooling. The following gives an overview:

Object-oriented Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits. Classes are extended by subclassing and mixin-based composition.

Functional It is also a functional language in the sense that every function is a value. Providing a lightweight syntax for defining anonymous functions, it supports higher-order functions, nested functions, and currying. With case classes and its built-in pattern matching, algebraic types used in many functional programming languages are supported.

Static type system The expressive type system enforces statically that abstractions are used in a safe and coherent manner. In particular, the type system supports: generic classes, variance annotations, upper and lower type bounds, inner classes and abstract types as object members, compound types, explicitly typed self references, implicit parameters and conversions, and polymorphic methods. The local type inference mechanism takes care that the user is not required to annotate the program with redundant type information.

Extensibility The development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it easy to add new language constructs in the form of libraries: Any method may be used as an infix or postfix operator. Closures are constructed automatically depending on the expected type. A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

Interoperability Scala is designed to inter-operate well with the JRE. Java features like annotations and Java generics have direct analogues in Scala. Those Scala features without Java analogues, such as default and named parameters, compile as close to Java as they can reasonably come. Scala has the same compilation model (separate compilation, dynamic class loading) like Java and allows access to existing libraries.

In the following, we give an overview of the most important Scala components, enabling the reader to understand the features of Scala used for the remaining parts of this thesis.

Scala has four types of **identifiers**. *Alphanumeric Identifiers* are composed of letters, underscores, and digits, beginning with a letter or underscore. The `$` counts as a letter, but is reserved for the use by Scala. It follows the camel case conventions of Java. The *Operator Identifiers* are composed of operator characters (most punctuation marks). This may not include: letters or digits, or `()[]{}'"_.,;,‘`. Operators that end in a colon `(:)` are right-associative, all others are left-associative. *Mixed Identifiers* are composed of an alphanumeric identifier, an underscore, and an operator identifier. Finally, *Literal Identifiers* are composed of any arbitrary string enclosed in backticks (the ‘ character).

All variables must be **declared**, with either `var` (if the value may change) or `val` (for constants). In Scala, `val` is preferred, unless there is a good reason to use `var`. If given an

initial value in the declaration, the variable's type is inferred and need not be explicitly stated. If explicitly stated, the type follows the variable and a colon, e.g., `var q: Boolean = true`. The types of variables must be declared when the variable is the formal parameter of a method, when the variable is declared but not initialized (this can only occur within a class), and when type parameters (generics) are enclosed in square brackets, e.g., `Array[Int]`. Within a method, variables must be given an initial value. Variables within a class (and not within a method) may optionally be given an initial value. When initial values are not given, `new` is required (e.g., `val ary = new Array[Int](5)`) and default values of `0`, `false`, or `null` are used. When initial values are given, `new` is not allowed (e.g., `val ary2 = Array(1, 2, 3)`). The type of a function is written with the double-headed arrow, =>. Except in the case of a single argument, parentheses are required when a function has no arguments (`() => return_type`), exactly one argument (`arg_type => return_type` or `(arg_type) => return_type`), and two or more arguments (`(arg_type_1, ..., arg_type_N) => return_type`).

Scala has no primitives for **data types**, only objects. Thus, the ladder of types looks like this:

- `Any`
  - `AnyVal`
    - `Boolean`, `Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double` (these are as in Java)
    - `Unit` (has only a single value, `()`; returned by functions that "do not return anything").
  - `AnyRef` (corresponds to `Object` in Java)
    - All `java.*` reference types
    - `ScalaObject`: Everything in `scala.*` references types, including arrays and lists.
  - `Null` (bottom of all `AnyRef` objects)
- `Nothing` (bottom of `Any`)

`Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double` are considered to be **numeric types**. The default type for integers is `Int`, and the default type for numbers is `Double`. The less accurate forms are converted to `Int` or `Double` when arithmetic is performed using them. All numbers are stored internally in a binary representation, but they may be written in various ways:

- Decimal integers use the digits `0` to `9`, with an optional + or - prefix.
- Decimal long integers are written as integers with an `l` or `L` suffix. Due to its resemblance to the digit `1`, use of the letter `l` is strongly discouraged.
- Octal integers use the digits `0` to `7`, beginning with a `0`, and an optional + or - prefix.
- Hexadecimal integers use the digits `0` to `9` and the letters `A` to `F` (may be lowercase), and an optional + or - prefix.
- Decimal double-precision numbers use the digits `0` to `9`, with an optional + or - prefix and an optional exponent. The exponent consists of an `e` or `E`, an optional sign, and one to three decimal digits.
- Decimal single-precision numbers are written as an integer or double-precision number with an `f` or `F` suffix.

Operations on numbers include + (addition), - (subtraction or negation), * (multiplication), / (division), and % (modulus). Operations on integers also include << (left shift), >> (right shift with sign extension), >>> (right shift with zero fill), & (bitwise and), | (bitwise or), and ^ (bitwise exclusive or).

A **String** may be enclosed in double quotes, or in triple double quotes. The latter is a raw string and may contain newlines. A `Symbol` is an interned string. A symbol literal starts with a single quote, `'`, followed by a letter, then zero or more letters and digits.

Lists are the most valuable data type in any functional language, including Scala. Lists are immutable and persistent. Operations that return lists do not modify the original list, but generally share structure with it. Additionally, they are homogeneous. All the elements of a list are the same type. The type of the empty list is `List[Nothing]`. When it is necessary to specify the type of a list, it is written as `List[Type]`. A literal list is written as `List(value, ..., value)`. Fundamental list methods are:

- `list.head` returns the first element in the list.
- `list.tail` returns the rest of the list, minus the head.
- `value::list` puts the value as the new head of the list (right-associative). The empty list is written as `List()` or as `Nil`. `List(1, 2, 3)` is equivalent to `1::2::3::Nil`.
- `list.isEmpty` checks if the list is empty.

Functions defined in terms of the above include:

- `list.length` returns the number of elements in the list.
- `list.last` returns the last element of the list.
- `list.init` returns a list with the last element removed.
- `list.take(n)` returns a list of the first `n` elements.
- `list.drop(n)` returns a list with the first `n` elements removed.
- `list1:::list2` appends the two lists.
- `list.reverse` returns a list in the reverse order.
- `list.splitAt(n)` returns the tuple (list take `n`, list drop `n`).
- `list1.zip(list2)` returns a list of tuples, where the first element of the tuple is taken from `list1` and the second from `list2`. The length of the result is the length of the shorter list.
- `list.mkString(string)` converts each element of the list into a string, and concatenates them with the string in between elements.
- `list.distinct` returns a list with duplicated elements removed.
- `listOfLists.flatten` takes a list of lists of elements and returns a list of elements.

The following **higher-order functions** are described as operations on lists, but most of them will operate on many types of sequences:

- `list.map(function)` returns a list in which the function of one argument has been applied to each element.
- `listOfLists.flatMap(function)` returns a list in which the function of one argument has been applied to each element of each sublist. Removes one level of nesting.

- `list.filter(predicate)` returns a list of the elements of the given list for which the predicate is true.
- `list.filterNot(predicate)` returns a list of the elements of the given list for which the predicate is false.
- `list.partition(predicate)` returns a tuple of two lists: a list of values that satisfy the predicate, and a list of those that do not.
- `list.foldLeft(value)(_ function _)` applies the function to each pair of elements of `value::list`, using each function result as the new first argument to the function, and returns the final value.
- `(value /: list)(_ function _)` is an alternate way of writing `foldLeft`.
- `list.foldRight(value)(_ function _)` applies the function to each pair of elements of the list with the value appended, starting from the right end of the list, using each function result as the new second argument to the function, and returns the final value.
- `(list : value)(_ function _)` is an alternate way of writing `foldRight`.
- `list.find(predicate)` returns the first value in the list that satisfies the predicate (as `Some(value)`), or `None` if no such value is found.
- `list.takeWhile(predicate)` returns a list of the values at the front of the given list that satisfy the predicate, stopping at the first value that does not.
- `list.dropWhile(predicate)` returns a list omitting those values at the front of the given list that satisfy the predicate.
- `list.span(predicate)` returns the tuple `(list takeWhile predicate, list dropWhile predicate)`.
- `list.forall(predicate)` checks if every element of the list satisfies the predicate.
- `list.exists(predicate)` checks if any element of the list satisfies the predicate.
- `list.sortWith(comparisonFunction)` sorts a list using the two-parameter comparison function. `list.sortWith(_ < _)` sorts the list in ascending order.
- `list.groupBy(function)` returns a `Map` of keys to values, where the keys are the results of applying the function to each list element, and the values are a `List` of values in `list` such that applying the function to that value yields that key.

A **tuple** consists of from 2 to 22 comma-separated values enclosed in parentheses. Tuples are immutable. To access the n-th value in a tuple `t`, the notation `t._n` is used, where `n` is a literal integer in the range 1 to 22. There may be spaces around the dot, but not between `_` and `n`.

Sets are immutable by default. Many additional operations are defined for sets, e.g., `contains`, `isEmpty`, `intersect`, `union`, and `diff`. Mutable sets may be imported from `scala.collection.mutable`.

Maps are immutable by default. A map is represented as a list of pairs that is, `Map((key, value), ..., (key, value))`. A more expressive syntax for writing the same thing is `Map(key -> value, ..., key -> value)`. The basic operations on maps are: i) Getting the value associated with a key. And ii), for mutable maps, associating a value with a key (mutable maps may be imported from `scala.collection.mutable`). Most list operations can also be used on maps. `map.get(key)` returns (as an `Option`) the value associated with the key. `map.getOrElse(key, default)` returns the value associated

with the key, or if the key is not in the map, the default value. map.put(key, value) for mutable maps only, associates the value with the key, and returns (as an Option) the old value.

Options are used when an operation may or may not succeed in returning a value. They are parameterized types, so one may have, e.g., an Option[*String*] type. The possible values are Some(value), where the value is of the correct type, or None, for the case where no value has been found. Although a few operations are defined for Option types, it is far more common to use a match expression to extract the value, if one exists.

The value of the if expression is the value of the **expression** that is chosen; or, if no condition is satisfied and there is no else clause, the value is the Unit value, ().

```
1 if (condition) expression
2 if (condition) expression else expression
3 if (condition) expression else if (condition) expression ...
4 if (condition) expression else if (condition) expression ... else expression
```

The for expression is used to create and return a sequence of results. The syntax is for (seq) yield expression, where:

- seq is a semicolon-separated sequence of generators, definitions, and filters, beginning with a generator.
  - A generator has the form variable <- list, where the list may be any expression resulting in a list.
  - A definition has the form variable = expression.
  - A filter has the form if condition.
- Where possible, the type of the result will be the same type as the seq.

The match expression uses pattern matching to select a case, then the value of the match expression is the value of the corresponding expression. The syntax is:

```
1 expression match {
2     case pattern1 => expression1
3     case pattern2 => expression2
4     ...
5     case patternN => expressionN
6 }
```

Patterns may be:

- A literal value of the same type as the expression, or a subtype of it.
- A variable, which will match any value (and may be used in the corresponding expression$_i$).
- An underscore, which will match anything. This is commonly used as the last catch-all pattern.
- A sequence, such as List(a, _, c), where the components are patterns.
- A tuple, where the components are patterns.
- An option type, Some(x) or None.
- A typed pattern, such as s:*String* or m:*Map*[_, _]. For parameterized types such as List and Map, specifying the types of their elements is not possible.
- The name of a case class, where patterns are used in place of the constructor parameters.

- A regular expression.
- Any of the above followed by a pattern guard (the word `if` followed by a `Boolean` expression).

Cases are tried in the order in which they occur. When a case is selected, the code for that case, and only for that one case, is executed.

Scala does not have **statements**, it has expressions, and every expression has a value. However, some expressions are executed purely for their side effects, and return the `Unit` value, `()`, which is essentially meaningless. Such expressions are basically the same as statements in other languages.

The syntax for **methods** is:

```
[override] [access] def methodName(arg: type, ..., arg: type) { body }
[override] [access] def methodName(arg: type, ..., arg: type): returnType = {
    body }
```

If overriding an inherited method, the keyword `override` is required. Access may be `private`, `protected`, or `public` by default. The type of each argument must be specified explicitly. The return type of a method can usually be omitted; but it must be declared when there is an explicit return statement, when the method is recursive, when the method calls another method with the same name, and when the inferred return type is more general than desired. For reasons of clarity, it is usually best to declare the return type. Braces around the body may be omitted if the body consists of a single expression, whose value is to be returned. Methods may be overloaded, as in Java. The parameters of a method or function are treated as if they were defined by `val`; they cannot be reassigned a new value.

The syntax for **function literals** is `(arg1: Type1, ..., argN: TypeN) => expression`. Within an enclosing argument list, the parentheses around the parameter list can usually be omitted. If each parameter is used only once, and the parameters appear in the expression in the same order as in the parameter list, the parameter list may be omitted, and underscores may be used in the expression, e.g., `"aBcDeF".map(_ toLower)`. Methods can often be passed as if they were functions, e.g, `"abc" map println` works. In other cases, the method name must be followed by an underscore to convert it to a partial function, e.g., `"abc" map (println _)`.

A **class** describes objects, as in Java. It may extend one other class, and it may include any number of traits. A class has a primary constructor, which is part of the class declaration itself.

`class Foo { ... }`
: By default, a class extends `AnyRef`.

`class Foo(name: String) { constructor_body }`
: The primary constructor is part of the class header. Parameters are put after the class name, and the constructor body is the entire class body.

`class Foo(a: Int, var b: Int, val c: Int) { ... }`
: Constructor parameters are handled as follows. Automatically, all parameters are automatically declared as private fields; a has private getters and setters; `var b` has public getters and setters; and `val c` has a public getter. A setter method for a parameter v has the name `v_=`.

`def this(parameters1) { this(parameters2); ... }`
: Auxiliary constructors are defined with `this`; the first statement in the auxiliary constructor must be a call to some other constructor.

`class Foo(a: Int , b: Int) extends Bar(a + b) { ... }`
When extending a class, any base class parameters are provided immediately.

`class Dog(name: String) extends Animal with Friend { ... }`
A comma-separated list of traits can follow the keyword `with`. Any uninitialized `var` and `val` defined by the trait must be initialized by the class.

A **case class** is defined by adding the keyword `case` before the word `class`. This results in additional methods being added to the class automatically by the compiler. It provides an `apply` method (allowing to omit the word `new` when constructing a new object of the class), as well as an `unapply` method (usage of case classes in pattern matching). Additionally, getters and setters for the constructor parameters are provided. Finally, it provides a nicer `toString`, == and `hashCode` methods, based on the constructor arguments.

In addition to creating objects from classes, you can declare **objects**. An object is declared like a class, but with the keyword `object` instead of `class`; also, an object cannot take parameters. Whereas a class declaration describes a blueprint for objects, an object declaration declares a single object (i.e., a singleton).

Finally, **traits** are declared like classes, used like Java interfaces, and may contain fully-defined methods. In the declaration of a class (or object), the syntax is:

```
1 // if the superclass is mentioned:
2 class ClassName extends SuperClass with Trait1, ..., TraitN
3 // otherwise:
4 class ClassName extends Trait1 with Trait2, ..., TraitN
```

C

ADDITIONAL INFORMATION

SCROLL can be found at:
`https://github.com/max-leuthaeuser/SCROLL`
(Accessed: 8th May 2017, 10.00)

The *SCROLLCompilerPlugin* can be found at:
`https://github.com/max-leuthaeuser/SCROLLCompilerPlugin`
(Accessed: 8th May 2017, 10.00)

D

SOURCE CODE

D.1. SOURCE CODE FOR THE ROBOTIC CO-WORKER

D.1.1. MACHINE.SCALA

Listing D.1.: Source code for the Machine.

```
// The player Machine, implementing the core behavior
case class Machine() {
  val ROTATION_PER_TICK = 40
  private var currentBox = Box()
  private var allowedToWork = true

  private def pickBox(speed: Int): Unit = {
    println(s"Picking($speed) Box(${currentBox.id})")
    Thread.sleep(1000)
  }

  private def handleBox(speed: Int): Unit = {
    println(s"Handling($speed) Box(${currentBox.id})")
    Thread.sleep(1000)
  }

  private def placeBox(speed: Int): Unit = {
    println(s"Placing($speed) Box(${currentBox.id})")
    currentBox = Box()
    Thread.sleep(1000)
  }

  def doWork(speed: Int): Unit = {
    pickBox(speed)
    handleBox(speed)
    placeBox(speed)
  }

  def isAllowedToWork: Boolean = allowedToWork

  def stop(): Unit = disallowToWork()

  def allowToWork(): Unit = { this.allowedToWork = true }

  def disallowToWork(): Unit = { this.allowedToWork = false }
}
```

D.1.2. HADDADINAUTOMATON.SCALA

Listing D.2.: Source code for HaddadinAutomaton.

```
object HaddadinAutomaton {
  // Events:
  case object E_collaborate extends RPAData
  case object E_confirm extends RPAData
  case object E_fault extends RPAData
  case object E_in_perception extends RPAData
  case object E_lost_perception extends RPAData
  case object E_out_perception extends RPAData
  case object E_stop_collaborate extends RPAData
  // States:
  case object Autonomous_Mode extends RPAState
  case object Collaborative_mode extends RPAState
  case object Fault_reaction_autonomous extends RPAState
  case object Fault_reaction_collaborative extends RPAState
  case object Fault_reaction_human extends RPAState
  case object Human_friendly_mode extends RPAState
}

class HaddadinAutomaton(val comp: HaddadinCompartment) extends
    RolePlayingAutomaton {
  import HaddadinAutomaton._
  // Player:
  val machine = comp.machine
  // Roles:
  val machine_autonomous = comp.machine_autonomous
  val machine_collaborative = comp.machine_collaborative
  val machine_faultreaction = comp.machine_faultreaction
  val machine_humanfriendly = comp.machine_humanfriendly

  when(Autonomous_Mode) {
    case Event(E_fault, _) => goto(Fault_reaction_autonomous)
    case Event(E_in_perception, _) => goto(Human_friendly_mode)
  }
  when(Collaborative_mode) {
    case Event(E_fault, _) => goto(Fault_reaction_collaborative)
    case Event(E_out_perception, _) => goto(Autonomous_Mode)
    case Event(E_stop_collaborate, _) => goto(Human_friendly_mode)
  }
  when(Fault_reaction_autonomous) {
    case Event(E_confirm, _) => goto(Autonomous_Mode)
  }
  when(Fault_reaction_collaborative) {
    case Event(E_confirm, _) => goto(Collaborative_mode)
  }
  when(Fault_reaction_human) {
    case Event(E_confirm, _) => goto(Human_friendly_mode)
  }
  when(Human_friendly_mode) {
    case Event(E_collaborate, _) => goto(Collaborative_mode)
    case Event(E_fault, _) => goto(Fault_reaction_human)
    case Event(E_out_perception, _) => goto(Autonomous_Mode)
  }
  when(Start) {
    case Event(Uninitialized, _) => goto(Autonomous_Mode)
  }
  whenUnhandled {
    case _ => stay()
  }
```

```
58
59     onTransition {
60       case Start -> Autonomous_Mode =>
61         comp.addPlaysRelation(machine, machine_autonomous)
62       case Autonomous_Mode -> Fault_reaction_autonomous =>
63         comp.removePlaysRelation(machine, machine_autonomous)
64         machine.stop()
65         comp.addPlaysRelation(machine, machine_faultreaction)
66       case Autonomous_Mode -> Human_friendly_mode =>
67         comp.removePlaysRelation(machine, machine_autonomous)
68         comp.addPlaysRelation(machine, machine_humanfriendly)
69       case Collaborative_mode -> Autonomous_Mode =>
70         comp.removePlaysRelation(machine, machine_collaborative)
71         comp.addPlaysRelation(machine, machine_autonomous)
72       case Collaborative_mode -> Fault_reaction_collaborative =>
73         comp.removePlaysRelation(machine, machine_collaborative)
74         machine.stop()
75         comp.addPlaysRelation(machine, machine_faultreaction)
76       case Collaborative_mode -> Human_friendly_mode =>
77         comp.removePlaysRelation(machine, machine_collaborative)
78         comp.addPlaysRelation(machine, machine_humanfriendly)
79       case Fault_reaction_autonomous -> Autonomous_Mode =>
80         comp.removePlaysRelation(machine, machine_faultreaction)
81         comp.addPlaysRelation(machine, machine_autonomous)
82       case Fault_reaction_collaborative -> Collaborative_mode =>
83         comp.removePlaysRelation(machine, machine_faultreaction)
84         comp.addPlaysRelation(machine, machine_collaborative)
85       case Fault_reaction_human -> Human_friendly_mode =>
86         comp.removePlaysRelation(machine, machine_faultreaction)
87         comp.addPlaysRelation(machine, machine_humanfriendly)
88       case Human_friendly_mode -> Autonomous_Mode =>
89         comp.removePlaysRelation(machine, machine_humanfriendly)
90         comp.addPlaysRelation(machine, machine_autonomous)
91       case Human_friendly_mode -> Collaborative_mode =>
92         comp.removePlaysRelation(machine, machine_humanfriendly)
93         comp.addPlaysRelation(machine, machine_collaborative)
94       case Human_friendly_mode -> Fault_reaction_human =>
95         comp.removePlaysRelation(machine, machine_humanfriendly)
96         machine.stop()
97         comp.addPlaysRelation(machine, machine_faultreaction)
98     }
99   }
```

D.1.3. HADDADINCOMPARTMENT.SCALA

Listing D.3.: Source code for HaddadinCompartment.

```
class HaddadinCompartment extends Compartment {
  // Player:
  val machine = Machine()
  // Roles:
  val machine_autonomous = Autonomous()
  val machine_collaborative = Collaborative()
  val machine_faultreaction = FaultReaction()
  val machine_humanfriendly = HumanFriendly()

  def getRotationPerTick: Int = (+machine).ROTATION_PER_TICK
  def isAllowedToWork: Boolean = (+machine).isAllowedToWork()
  def doWork(): Unit = (+machine).doWork(getRotationPerTick)
  def main(): Unit = { Entry().main(machine) }

  case class Entry() {
    def main(machine: Machine): Unit = {
      while (true) { (+machine).doWork((+machine).ROTATION_PER_TICK) }
    }
  }
  case class Autonomous() {
    private val ROTATION_PER_TICK = 40
    def doWork(speed: Int): Unit = {
      implicit val dd = Bypassing(_.isInstanceOf[Autonomous])
      (+this).doWork(ROTATION_PER_TICK)
    }
  }
  case class Collaborative() {
    private val ROTATION_PER_TICK = 5
    def doWork(speed: Int): Unit = {
      implicit val dd = Bypassing(_.isInstanceOf[Collaborative])
      (+this).doWork(ROTATION_PER_TICK)
    }
  }
  case class HumanFriendly() {
    private val ROTATION_PER_TICK = 20
    def doWork(speed: Int): Unit = {
      implicit val dd = Bypassing(_.isInstanceOf[HumanFriendly])
      (+this).doWork(ROTATION_PER_TICK)
    }
  }
  case class FaultReaction() {
    private val ROTATION_PER_TICK = 0
    def doWork(speed: Int): Unit = { // we do nothing }
    def isAllowedToWork: Boolean = false
  }
}
```

D.1.4. SMARTSENSORS.SCALA

Listing D.4.: Source code for SmartSensors.

```
// Steering the sensor based adaptation
class SmartSensors(comp: HaddadinCompartment) {
  val automaton = new HaddadinAutomaton(comp)

  def workerIsIn(room: Room): Boolean = worker.isIn(room)
  def inCollaboration: Boolean = workerIsIn(machineArea)
  def outOfContext: Boolean = workerIsIn(roomA)  workerIsIn(exit)
  def inContext: Boolean = !outOfContext
  def collabStart: Boolean = inCollaboration
  def collabStop: Boolean = !inCollaboration
  def fault: Boolean = workerIsIn(machine)
  def confirm: Boolean = !fault

  val constraintsActor = new Actor {
    def receive = {
      case Check =>
        if (fault) {
          automaton ! E_fault
        }
        if (confirm) {
          comp.machine.allowToWork()
          automaton ! E_confirm
        }
        if (outOfContext) {
          automaton ! E_out_perception
        }
        if (inContext) {
          automaton ! E_in_perception
        }
        if (collabStart) {
          automaton ! E_collaborate
        }
        if (collabStop) {
          automaton ! E_stop_collaborate
        }
    }
  }

  def start(): Unit = {
    schedule(0 seconds, 100 milliseconds, constraintsActor, Check)
    automaton ! Uninitialized
  }
}
```

D.1.5. HADDADINDEMO.SCALA

Listing D.5.: Source code for `HaddadinDemo`.

```
1  // The main application entry point starting the actual simulation
2  object HaddadinDemo {
3    def main(args: Array[String]): Unit = {
4      val comp = new HaddadinCompartment()
5      new SmartSensors(comp).start()
6      comp.main()
7    }
8  }
```

D.2. SOURCE CODE FOR *SCROLL*

D.2.1. COMPARTMENT.SCALA

Listing D.6.: Source code for `Compartment.scala`.

```
package scroll.internal

import java.lang.reflect.Method

import scroll.internal.errors.SCROLLErrors._
import scroll.internal.support._
import UnionTypes.RoleUnionTypes
import scroll.internal.graph.CachedScalaRoleGraph
import scroll.internal.util.ReflectiveHelper

import scala.util.{Failure, Success, Try}
import scala.annotation.tailrec
import scala.reflect.{ClassTag, classTag}

/**
  * This Trait allows for implementing an objectified collaboration with a limited number of participating roles and a fixed scope.
  *
  * ==Overview==
  * Roles are dependent on some sort of context. We call them compartments. A typical example of a compartment is a university,
  * which contains the roles Student and Teacher collaborating in Courses. Everything in SCROLL happens inside of Compartments
  * but roles (implemented as standard Scala classes) can be defined or imported from everywhere. Just mix in this Trait
  * into your own specific compartment class or create an anonymous instance.
  *
  * ==Example==
  * {{{
  * val player = new Player()
  * new Compartment {
  *   class RoleA
  *   class RoleB
  *
  *   player play new RoleA()
  *   player play new RoleB()
  *
  *   // call some behaviour
  * }
  * }}}
  */
trait Compartment
  extends RoleConstraints
    with RoleRestrictions
    with RoleGroups
    with Relationships
    with QueryStrategies
    with RoleUnionTypes {

  protected val plays = new CachedScalaRoleGraph()

  implicit def either2TorException[T](either: Either[_, T]): T = either.fold(
    l => {
      throw new RuntimeException(l.toString)
    }, r => {
      r
```

```
53      })
54  
55    /**
56      * Declaring a is-part-of relation between compartments.
57      */
58    def partOf(other: Compartment): Unit = {
59      require(null != other)
60      plays.merge(other.plays)
61    }
62  
63    /**
64      * Declaring a bidirectional is-part-of relation between compartment.
65      */
66    def union(other: Compartment): Compartment = {
67      require(null != other)
68      other.partOf(this)
69      this.partOf(other)
70      this
71    }
72  
73    /**
74      * Removing is-part-of relation between compartments.
75      */
76    def notPartOf(other: Compartment): Unit = {
77      require(null != other)
78      plays.detach(other.plays)
79    }
80  
81    /**
82      * Query the role playing graph for all player instances that do conform
        to the given matcher.
83      *
84      * @param matcher the matcher that should match the queried player
        instance in the role playing graph
85      * @tparam T the type of the player instance to query for
86      * @return all player instances as Seq, that do conform to the given matcher
87      */
88    def all[T: ClassTag](matcher: RoleQueryStrategy = MatchAny()): Seq[T] = {
89  
        plays.allPlayers.filter(ReflectiveHelper.is[T]).map(_.asInstanceOf[T]).filter(a
        => {
90        getCoreFor(a) match {
91          case p :: Nil => matcher.matches(p)
92          case Nil => false
93          case l => l.forall(matcher.matches)
94        }
95      })
96    }
97  
98    /**
99      * Query the role playing graph for all player instances that do conform
        to the given function.
100     *
101     * @param matcher the matching function that should match the queried
        player instance in the role playing graph
102     * @tparam T the type of the player instance to query for
103     * @return all player instances as Seq, that do conform to the given matcher
104     */
105   def all[T: ClassTag](matcher: T => Boolean): Seq[T] =
106 
        plays.allPlayers.filter(ReflectiveHelper.is[T]).map(_.asInstanceOf[T]).filter(a
        => {
```

```
107        getCoreFor(a) match {
108          case p :: Nil => matcher(p.asInstanceOf[T])
109          case Nil => false
110          case l: Seq[Any] => l.forall(i => matcher(i.asInstanceOf[T]))
111        }
112      })
113
114    private def safeReturn[T](seq: Seq[T], typeName: String):
         Either[TypeError, Seq[T]] = seq match {
115      case Nil => Left(TypeNotFound(typeName))
116      case s => Right(s)
117    }
118
119    /**
120      * Query the role playing graph for all player instances that do conform
       to the given matcher and return the first found.
121      *
122      * @param matcher the matcher that should match the queried player
       instance in the role playing graph
123      * @tparam T the type of the player instance to query for
124      * @return the first player instance, that does conform to the given
       matcher or an appropriate error
125      */
126    def one[T: ClassTag](matcher: RoleQueryStrategy = MatchAny()):
         Either[TypeError, T] = safeReturn(all[T](matcher),
         classTag[T].toString).fold(
127      l => {
128        Left(l)
129      }, r => {
130        Right(r.head)
131      })
132
133
134    /**
135      * Query the role playing graph for all player instances that do conform
       to the given function and return the first found.
136      *
137      * @param matcher the matching function that should match the queried
       player instance in the role playing graph
138      * @tparam T the type of the player instance to query for
139      * @return the first player instances, that do conform to the given
       matcher or an appropriate error
140      */
141    def one[T: ClassTag](matcher: T => Boolean): Either[TypeError, T] =
         safeReturn(all[T](matcher), classTag[T].toString).fold(
142      l => {
143        Left(l)
144      }, r => {
145        Right(r.head)
146      })
147
148    /**
149      * Adds a play relation between core and role.
150      *
151      * @tparam C type of core
152      * @tparam R type of role
153      * @param core the core to add the given role at
154      * @param role the role that should added to the given core
155      */
156    def addPlaysRelation[C <: AnyRef : ClassTag, R <: AnyRef :
         ClassTag](core: C, role: R): Unit = {
157      require(null != core)
```

```
158     require(null != role)
159     validate(core, role)
160     plays.addBinding(core, role)
161   }
162 
163   /**
164     * Removes the play relation between core and role.
165     *
166     * @tparam C type of core
167     * @tparam R type of role
168     * @param core the core the given role should removed from
169     * @param role the role that should removed from the given core
170     */
171   def removePlaysRelation[C <: AnyRef : ClassTag, R <: AnyRef :
        ClassTag](core: C, role: R): Unit = {
172     require(null != core)
173     require(null != role)
174     plays.removeBinding(core, role)
175   }
176 
177   /**
178     * Transfers a role from one core to another.
179     *
180     * @tparam F type of core the given role should be removed from
181     * @tparam T type of core the given role should be attached to
182     * @tparam R type of role
183     * @param coreFrom the core the given role should be removed from
184     * @param coreTo   the core the given role should be attached to
185     * @param role     the role that should be transferred
186     */
187   def transferRole[F <: AnyRef : ClassTag, T <: AnyRef : ClassTag, R <:
        AnyRef : ClassTag](coreFrom: F, coreTo: T, role: R): Unit = {
188     require(null != coreFrom)
189     require(null != coreTo)
190     require(coreFrom != coreTo, "You can not transfer a role from itself.")
191     removePlaysRelation(coreFrom, role)
192     addPlaysRelation(coreTo, role)
193   }
194 
195   @tailrec
196   private def getCoreFor(role: Any): Seq[Any] = {
197     require(null != role)
198     role match {
199       case cur: Player[_] => getCoreFor(cur.wrapped)
200       case cur: Any if plays.containsPlayer(cur) =>
201         val r = plays.getPredecessors(cur)
202         if (r.isEmpty) {
203           Seq(cur)
204         } else {
205           if (r.size == 1) {
206             getCoreFor(r.head)
207           } else {
208             r
209           }
210         }
211       case _ => Seq(role)
212     }
213   }
214 
215   /**
216     * Generic Trait that enables dynamic invocation of role methods that are
      not natively available on the player object.
```

```
217     */
218   trait SCROLLDynamic extends Dynamic {
219     /**
220       * Allows to call a function with arguments.
221       *
222       * @param name          the function name
223       * @param args          the arguments handed over to the given function
224       * @param dispatchQuery the dispatch rules that should be applied
225       * @tparam E return type
226       * @tparam A argument type
227       * @return the result of the function call or an appropriate error
228       */
229     def applyDynamic[E, A](name: String)(args: A*)(implicit dispatchQuery:
          DispatchQuery = DispatchQuery.empty): Either[SCROLLError, E]
230
231     /**
232       * Allows to call a function with named arguments.
233       *
234       * @param name          the function name
235       * @param args          tuple with the the name and argument handed over
          to the given function
236       * @param dispatchQuery the dispatch rules that should be applied
237       * @tparam E return type
238       * @return the result of the function call or an appropriate error
239       */
240     def applyDynamicNamed[E](name: String)(args: (String, Any)*)(implicit
          dispatchQuery: DispatchQuery = DispatchQuery.empty):
          Either[SCROLLError, E]
241
242     /**
243       * Allows to read a field.
244       *
245       * @param name          of the field
246       * @param dispatchQuery the dispatch rules that should be applied
247       * @tparam E return type
248       * @return the result of the field access or an appropriate error
249       */
250     def selectDynamic[E](name: String)(implicit dispatchQuery: DispatchQuery
          = DispatchQuery.empty): Either[SCROLLError, E]
251
252     /**
253       * Allows to write field updates.
254       *
255       * @param name          of the field
256       * @param value         the new value to write
257       * @param dispatchQuery the dispatch rules that should be applied
258       */
259     def updateDynamic(name: String)(value: Any)(implicit dispatchQuery:
          DispatchQuery = DispatchQuery.empty): Unit
260   }
261
262   trait Dispatchable {
263     /**
264       * For empty argument list dispatch.
265       *
266       * @param on the instance to dispatch the given method m on
267       * @param m  the method to dispatch
268       * @tparam E the return type of method m
269       * @return the resulting return value of the method invocation or an
          appropriate error
270       */
271     def dispatch[E](on: Any, m: Method): Either[InvocationError, E]
```

```
272
273      /**
274        * For multi-argument dispatch.
275        *
276        * @param on   the instance to dispatch the given method m on
277        * @param m    the method to dispatch
278        * @param args the arguments to pass to method m
279        * @tparam E the return type of method m
280        * @tparam A the type of the argument values
281        * @return the resulting return value of the method invocation or an
           appropriate error
282        */
283      def dispatch[E, A](on: Any, m: Method, args: Seq[A]):
             Either[InvocationError, E]
284    }
285
286    /**
287      * Trait handling the actual dispatching of role methods.
288      */
289    trait SCROLLDispatch extends Dispatchable {
290      override def dispatch[E](on: Any, m: Method): Either[InvocationError, E] = {
291        require(null != on)
292        require(null != m)
293        Try(ReflectiveHelper.resultOf[E](on, m)) match {
294          case Success(s) => Right(s)
295          case Failure(_) =>
               Left(IllegalRoleInvocationSingleDispatch(on.toString, m.getName))
296        }
297      }
298
299      override def dispatch[E, A](on: Any, m: Method, args: Seq[A]):
             Either[InvocationError, E] = {
300        require(null != on)
301        require(null != m)
302        require(null != args)
303        Try(ReflectiveHelper.resultOf[E](on, m,
              args.map(_.asInstanceOf[Object]))) match {
304          case Success(s) => Right(s)
305          case Failure(_) =>
               Left(IllegalRoleInvocationMultipleDispatch(on.toString, m.getName,
               args.toString()))
306        }
307      }
308
309    }
310
311    /**
312      * Explicit helper factory method for creating a new Player instance
313      * without the need to relying on the implicit mechanics of Scala.
314      *
315      * @param obj the player or role that is wrapped into this dynamic player type
316      * @return a new Player instance wrapping the given object
317      */
318    def newPlayer(obj: Object): Player[Object] = new Player(obj)
319
320    /**
321      * Implicit wrapper class to add basic functionality to roles and its
         players as unified types.
322      *
323      * @param wrapped the player or role that is wrapped into this dynamic type
324      * @tparam T type of wrapped object
325      */
```

```
326   implicit class Player[T <: AnyRef : ClassTag](val wrapped: T) extends
        SCROLLDynamic with SCROLLDispatch {
327     /**
328       * Applies lifting to Player
329       *
330       * @return an lifted Player instance with the calling object as wrapped.
331       */
332     def unary_+ : Player[T] = this
333
334     /**
335       * Returns the player of this player instance if this is a role, or this
        itself.
336       *
337       * @param dispatchQuery provide this to sort the resulting instances if
        a role instance is played by multiple core objects
338       * @return the player of this player instance if this is a role, or this
        itself or an appropriate error
339       */
340     def player(implicit dispatchQuery: DispatchQuery = DispatchQuery.empty):
          Either[TypeError, Any] = dispatchQuery.filter(getCoreFor(this))
          match {
341       case elem :: Nil => Right(elem)
342       case l: Seq[T] => Right(l.head)
343       case _ => Left(TypeNotFound(this.getClass.toString))
344     }
345
346     /**
347       * Adds a play relation between core and role.
348       *
349       * @tparam R type of role
350       * @param role the role that should be played
351       * @return this
352       */
353     def play[R <: AnyRef : ClassTag](role: R): Player[T] = {
354       wrapped match {
355         case p: Player[_] => addPlaysRelation[T,
              R](p.wrapped.asInstanceOf[T], role)
356         case p: Any => addPlaysRelation[T, R](p.asInstanceOf[T], role)
357         case _ => // do nothing
358       }
359       this
360     }
361
362     /**
363       * Alias for [[Player.play]].
364       *
365       * @tparam R type of role
366       * @param role the role that should be played
367       * @return this
368       */
369     def <+>[R <: AnyRef : ClassTag](role: R): Player[T] = play(role)
370
371     /**
372       * Adds a play relation between core and role but always returns the
        player instance.
373       *
374       * @tparam R type of role
375       * @param role the role that should played
376       * @return the player instance
377       */
378     def playing[R <: AnyRef : ClassTag](role: R): T = play(role).wrapped
379
```

```
380     /**
381       * Alias for [[Player.playing]].
382       *
383       * @tparam R type of role
384       * @param role the role that should played
385       * @return the player instance
386       */
387     def <=>[R <: AnyRef : ClassTag](role: R): T = playing(role)
388 
389     /**
390       * Removes the play relation between core and role.
391       *
392       * @param role the role that should be removed
393       * @return this
394       */
395     def drop[R <: AnyRef : ClassTag](role: R): Player[T] = {
396       removePlaysRelation[T, R](wrapped, role)
397       this
398     }
399 
400     /**
401       * Alias for [[Player.drop]].
402       *
403       * @param role the role that should be removed
404       * @return this
405       */
406     def <->[R <: AnyRef : ClassTag](role: R): Player[T] = drop(role)
407 
408     /**
409       * Transfers a role to another player.
410       *
411       * @tparam R type of role
412       * @param role the role to transfer
413       */
414     def transfer[R <: AnyRef : ClassTag](role: R) = new {
415       def to[P <: AnyRef : ClassTag](player: P): Unit = {
416         transferRole[T, P, R](wrapped, player, role)
417       }
418     }
419 
420     /**
421       * Checks of this Player is playing a role of the given type.
422       */
423     def isPlaying[E: ClassTag]: Boolean =
          plays.getRoles(wrapped).exists(ReflectiveHelper.is[E])
424 
425     /**
426       * Checks of this Player has an extension of the given type.
427       * Alias for [[Player.isPlaying]].
428       */
429     def hasExtension[E: ClassTag]: Boolean = isPlaying[E]
430 
431     override def applyDynamic[E, A](name: String)(args: A*)(implicit
          dispatchQuery: DispatchQuery = DispatchQuery.empty):
          Either[SCROLLError, E] = {
432       val core = dispatchQuery.filter(getCoreFor(wrapped)).head
433       val anys = dispatchQuery.filter(Seq(core, wrapped) ++ plays.getRoles(core))
434       anys.foreach(r => {
435         ReflectiveHelper.findMethod(r, name, args.toSeq).foreach(fm => {
436           args match {
437             case Nil => return dispatch(r, fm)
438             case _ => return dispatch(r, fm, args.toSeq)
```

```
439           }
440         })
441       })
442       // otherwise give up
443       Left(RoleNotFound(core.toString, name, args.toString()))
444     }
445
446     override def applyDynamicNamed[E](name: String)(args: (String,
        ↪ Any)*)(implicit dispatchQuery: DispatchQuery = DispatchQuery.empty):
        ↪ Either[SCROLLError, E] =
447       applyDynamic(name)(args.map(_._2): _*)(dispatchQuery)
448
449     override def selectDynamic[E](name: String)(implicit dispatchQuery:
        ↪ DispatchQuery = DispatchQuery.empty): Either[SCROLLError, E] = {
450       val core = dispatchQuery.filter(getCoreFor(wrapped)).head
451       val anys = dispatchQuery.filter(Seq(core, wrapped) ++ plays.getRoles(core))
452       anys.find(ReflectiveHelper.hasMember(_, name)) match {
453         case Some(r) => Right(ReflectiveHelper.propertyOf(r, name))
454         case None => Left(RoleNotFound(core.toString, name, ""))
455       }
456     }
457
458     override def updateDynamic(name: String)(value: Any)(implicit
        ↪ dispatchQuery: DispatchQuery = DispatchQuery.empty): Unit = {
459       val core = dispatchQuery.filter(getCoreFor(wrapped)).head
460       val anys = dispatchQuery.filter(Seq(core, wrapped) ++ plays.getRoles(core))
461       anys.find(ReflectiveHelper.hasMember(_, name)) match {
462         case Some(r) => ReflectiveHelper.setPropertyOf(r, name, value)
463         case None => // do nothing
464       }
465     }
466
467     override def equals(o: Any): Boolean = o match {
468       case other: Player[_] => getCoreFor(wrapped) == getCoreFor(other.wrapped)
469       case other: Any => getCoreFor(wrapped) match {
470         case Nil => false
471         case p :: Nil => p == other
472         case _ => false
473       }
474       case _ => false // default case
475     }
476
477     override def hashCode(): Int = wrapped.hashCode()
478   }
479
480 }
```

D.2.2. DISPATCHQUERY.SCALA

Listing D.7.: Source code for DispatchQuery.scala.

```scala
package scroll.internal.support

import scroll.internal.support.DispatchQuery._

/**
 * Companion object for [[scroll.internal.support.DispatchQuery]] providing
 * some static dispatch functions and a fluent dispatch query creation API.
 */
object DispatchQuery {

  /**
   * Use this in [[DispatchQuery.sortedWith]] to state that no sorting between the objects in comparison should happen.
   */
  val identity: Boolean = false

  /**
   * Use this in [[DispatchQuery.sortedWith]] to state that always swapping between the objects in comparison should happen.
   */
  val swap: Boolean = true

  /**
   * Function to use in [[DispatchQuery.sortedWith]] to simply reverse the set of resulting edges.
   */
  val reverse: PartialFunction[(Any, Any), Boolean] = {
    case (_, _) => swap
  }

  /**
   * Function always returning true
   */
  val anything: Any => Boolean = _ => true
  /**
   * Function always returning false
   */
  val nothing: Any => Boolean = _ => false

  def From(f: Any => Boolean) = new {
    def To(t: Any => Boolean) = new {
      def Through(th: Any => Boolean) = new {
        def Bypassing(b: Any => Boolean): DispatchQuery =
          new DispatchQuery(new From(f), new To(t), new Through(th), new Bypassing(b))
      }
    }
  }

  def Bypassing(b: Any => Boolean): DispatchQuery =
    new DispatchQuery(new From(anything, empty = true), new To(anything, empty = true), new Through(anything, empty = true), new Bypassing(b))

  def empty: DispatchQuery = new DispatchQuery(new From(anything), new To(anything), new Through(anything), new Bypassing(nothing), empty = true)

```

```
51   /**
52     * Dispatch filter selecting the sub-path from the starting edge until the end
53     * of the path given as Seq, w.r.t. the evaluation of the selection function.
54     *
55     * @param sel   the selection function to evaluate on each element of the path
56     * @param empty if set to true, the path will be returned unmodified
57     */
58   private class From(val sel: Any => Boolean, empty: Boolean = false)
       extends (Seq[Any] => Seq[Any]) {
59     override def apply(edges: Seq[Any]): Seq[Any] = if (empty) {
60       edges
61     } else {
62       edges.slice(edges.indexWhere(sel), edges.size)
63     }
64   }
65
66   /**
67     * Dispatch filter selecting the sub-path from the last edge until the end
68     * of the path given as Seq, w.r.t. the evaluation of the selection function.
69     *
70     * @param sel   the selection function to evaluate on each element of the path
71     * @param empty if set to true, the path will be returned unmodified
72     */
73   private class To(val sel: Any => Boolean, empty: Boolean = false) extends
       (Seq[Any] => Seq[Any]) {
74     override def apply(edges: Seq[Any]): Seq[Any] = if (empty) {
75       edges
76     } else {
77       edges.lastIndexWhere(sel) match {
78         case -1 => edges
79         case _ => edges.slice(0, edges.lastIndexWhere(sel) + 1)
80       }
81     }
82   }
83
84   /**
85     * Dispatch filter to specify which edges to keep on the path given as Seq,
86     * w.r.t. the evaluation of the selection function.
87     *
88     * @param sel   the selection function to evaluate on each element of the path
89     * @param empty if set to true, the path will be returned unmodified
90     */
91   private class Through(sel: Any => Boolean, empty: Boolean = false) extends
       (Seq[Any] => Seq[Any]) {
92     override def apply(edges: Seq[Any]): Seq[Any] = if (empty) {
93       edges
94     } else {
95       edges.filter(sel)
96     }
97   }
98
99   /**
100    * Dispatch filter to specify which edges to skip on the path given as Seq,
101    * w.r.t. the evaluation of the selection function.
102    *
103    * @param sel   the selection function to evaluate on each element of the path
104    * @param empty if set to true, the path will be returned unmodified
105    */
106  private class Bypassing(sel: Any => Boolean, empty: Boolean = false)
       extends (Seq[Any] => Seq[Any]) {
107    override def apply(edges: Seq[Any]): Seq[Any] = if (empty) {
108      edges
```

```
109      } else {
110        edges.filterNot(sel)
111      }
112    }
113
114  }
115
116  /**
117    * Composed dispatch query, i.e. applying the composition of all dispatch
       queries the given set of edges
118    * through the function [[DispatchQuery.filter]].
119    * All provided queries must be side-effect free!
120    *
121    * @param from      query selecting the starting element for the role
       dispatch query
122    * @param to        query selecting the end element for the role dispatch query
123    * @param through   query specifying intermediate elements for the role
       dispatch query
124    * @param bypassing query specifying all elements to be left out for the
       role dispatch query
125    */
126  class DispatchQuery(
127                       from: From,
128                       to: To,
129                       through: Through,
130                       bypassing: Bypassing,
131                       private val empty: Boolean = false,
132                       private var _sortedWith: Option[(Any, Any) => Boolean]
                            = Option.empty
133                     ) {
134    def isEmpty: Boolean = empty
135
136    /**
137      * Set the function to later sort all dynamic extensions during
         [[DispatchQuery.filter]].
138      *
139      * @param f the sorting function
140      * @return this
141      */
142    def sortedWith(f: PartialFunction[(Any, Any), Boolean]): DispatchQuery = {
143      _sortedWith = Some({ case (a, b) => f.applyOrElse((a, b), (_: (Any, Any))
           => identity) })
144      this
145    }
146
147    def filter(anys: Seq[Any]): Seq[Any] = {
148      val r = if (isEmpty) {
149        anys.distinct.reverse
150      } else {
151        from.andThen(to).andThen(through).andThen(bypassing)(anys.distinct).reverse
152      }
153      _sortedWith match {
154        case Some(f) => r.sortWith(f)
155        case None => r
156      }
157    }
158  }
```

D.2.3. QUERYSTRATEGIES.SCALA

Listing D.8.: Source code for `QueryStrategies.scala`.

```
1  package scroll.internal.support
2
3  import scroll.internal.util.ReflectiveHelper
4
5  /**
6    * Allows to write queries looking for the content of an attribute of the
     certain role playing
7    * object or the return value of one of its functions.
8    */
9  trait QueryStrategies {
10
11   implicit class RoleQueryStrategy(name: String) {
12     def matches(on: Any): Boolean = true
13
14     /**
15       * Returns the value the queried attribute.
16       *
17       * @param value the name of the attribute that is queried
18       * @tparam T its type
19       * @return the value of the queried attribute
20       */
21     def ==#[T](value: T): WithProperty[T] = WithProperty(name, value)
22
23     /**
24       * Returns the return value the queried function.
25       *
26       * @param value the name of the function that is queried
27       * @tparam T its return type
28       * @return the return value of the queried function
29       */
30     def ==>[T](value: T): WithResult[T] = WithResult(name, value)
31   }
32
33   case class MatchAny() extends RoleQueryStrategy("")
34
35   case class WithProperty[T](name: String, value: T) extends
       RoleQueryStrategy(name) {
36     override def matches(on: Any): Boolean =
         ReflectiveHelper.propertyOf[T](on, name) == value
37   }
38
39   case class WithResult[T](name: String, result: T) extends
       RoleQueryStrategy(name) {
40     override def matches(on: Any): Boolean = ReflectiveHelper.resultOf[T](on,
         name) == result
41   }
42
43 }
```

D.2.4. RELATIONSHIPS.SCALA

Listing D.9.: Source code for Relationships.scala.

```scala
package scroll.internal.support

import scroll.internal.Compartment
import scroll.internal.util.Many
import scala.reflect.ClassTag

/**
  * Allows to add and check role relationships to a compartment instance.
  */
trait Relationships {
  self: Compartment =>

  import Relationship._

  /**
    * Companion object for
    [[scroll.internal.support.Relationships.Relationship]] providing
    * some predefined multiplicities and a fluent relationship creation API.
    */
  object Relationship {

    sealed trait Multiplicity

    trait ExpMultiplicity extends Multiplicity

    case class MMany() extends ExpMultiplicity

    case class ConcreteValue(v: Ordered[Int]) extends ExpMultiplicity {
      require(v >= 0)

      def To(t: ExpMultiplicity): Multiplicity = RangeMultiplicity(v, t)
    }

    implicit def orderedToConcreteValue(v: Ordered[Int]): ExpMultiplicity = v
      match {
      case Many() => MMany()
      case _ => ConcreteValue(v)
    }

    implicit def intToConcreteValue(v: Int): ConcreteValue = ConcreteValue(v)

    case class RangeMultiplicity(from: ExpMultiplicity, to: ExpMultiplicity)
      extends Multiplicity

    def apply(name: String) = new {
      def from[L : ClassTag](leftMul: Multiplicity) = new {
        def to[R : ClassTag](rightMul: Multiplicity): Relationship[L, R] =
          new Relationship(name, leftMul, rightMul)
      }
    }

  }

  /**
    * Class representation of a relationship between two (role) types.
    *
    * @param name     name of the relationship
    * @param leftMul  multiplicity of the left side of the relationship
```

```
55     * @param rightMul multiplicity of the right side of the relationship
56     * @tparam L type of the role of the left side of the relationship
57     * @tparam R type of the role of the right side of the relationship
58     */
59   class Relationship[L : ClassTag, R : ClassTag](name: String,
60                                                   var leftMul: Multiplicity,
61                                                   var rightMul: Multiplicity) {
62 
63     private def checkMul[T](m: Multiplicity, on: Seq[T]): Seq[T] = {
64       m match {
65         case MMany() => assert(on.nonEmpty, s"With left multiplicity for '$name' of '*', the resulting role set should not be empty!")
66         case ConcreteValue(v) => assert(v.compare(on.size) == 0, s"With a concrete multiplicity for '$name' of '$v' the resulting role set should have the same size!")
67         case RangeMultiplicity(f, t) => (f, t) match {
68           case (ConcreteValue(v1), ConcreteValue(v2)) => assert(v1 <= on.size && v2 >= on.size, s"With a multiplicity for '$name' from '$v1' to '$v2', the resulting role set size should be in between!")
69           case (ConcreteValue(v), MMany()) => assert(v <= on.size, s"With a multiplicity for '$name' from '$v' to '*', the resulting role set size should be in between!")
70           case _ => throw new RuntimeException("This multiplicity is not allowed!") // default case
71         }
72         case _ => throw new RuntimeException("This multiplicity is not allowed!") // default case
73       }
74       on
75     }
76 
77     /**
78       * Get all instances of the left side of the relationship w.r.t. the provided matching function and checking the multiplicity.
79       *
80       * @param matcher a matching function to select the appropriate instances
81       * @return all instances of the left side of the relationship w.r.t. the provided matching function.
82       */
83     def left(matcher: L => Boolean = _ => true): Seq[L] = checkMul(leftMul, all[L](matcher))
84 
85     /**
86       * Get all instances of the right side of the relationship w.r.t. the provided matching function and checking the multiplicity.
87       *
88       * @param matcher a matching function to select the appropriate instances
89       * @return all instances of the right side of the relationship w.r.t. the provided matching function.
90       */
91     def right(matcher: R => Boolean = _ => true): Seq[R] = checkMul(rightMul, all[R](matcher))
92 
93   }
94 
95 }
```

D.2.5. ROLECONSTRAINTS.SCALA

Listing D.10.: Source code for RoleConstraints.scala.

```scala
package scroll.internal.support

import scroll.internal.Compartment
import scroll.internal.util.ReflectiveHelper

import scala.reflect.{ClassTag, classTag}
import com.google.common.graph.{GraphBuilder, Graphs, MutableGraph}

import scala.collection.JavaConverters._

/**
  * Allows to add and check role constraints (Riehle constraints) to a compartment instance.
  */
trait RoleConstraints {
  self: Compartment =>

  protected val roleImplications: MutableGraph[String] = GraphBuilder.directed().build[String]()
  protected val roleEquivalents: MutableGraph[String] = GraphBuilder.directed().build[String]()
  protected val roleProhibitions: MutableGraph[String] = GraphBuilder.directed().build[String]()

  private def isInstanceOf(mani: String, that: Any) =
    ReflectiveHelper.simpleName(that.getClass.toString) == ReflectiveHelper.simpleName(mani)

  private def checkImplications(player: Any, role: Any): Unit = {
    roleImplications.nodes().asScala.filter(isInstanceOf(_, role)).toList match {
      case Nil => //done, thanks
      case list =>
        val allImplicitRoles = list.flatMap(Graphs.reachableNodes(roleImplications, _).asScala)
        val allRoles = plays.getRoles(player).diff(Seq(player))
        allImplicitRoles.foreach(r => if (!allRoles.exists(isInstanceOf(r, _))) {
          throw new RuntimeException(s"Role implication constraint violation: '$player' should play role '$r', but it does not!")
        })
    }
  }

  private def checkEquivalence(player: Any, role: Any): Unit = {
    roleEquivalents.nodes().asScala.filter(isInstanceOf(_, role)).toList match {
      case Nil => //done, thanks
      case list =>
        val allEquivalentRoles = list.flatMap(Graphs.reachableNodes(roleEquivalents, _).asScala)
        val allRoles = plays.getRoles(player).diff(Seq(player))
        allEquivalentRoles.foreach(r => if (!allRoles.exists(isInstanceOf(r, _))) {
          throw new RuntimeException(s"Role equivalence constraint violation: '$player' should play role '$r', but it does not!")
        })
    }
  }

  private def checkProhibitions(player: Any, role: Any): Unit = {
```

```
49        roleProhibitions.nodes().asScala.filter(isInstanceOf(_, role)).toList match {
50          case Nil => //done, thanks
51          case list =>
52            val allProhibitedRoles =
                list.flatMap(Graphs.reachableNodes(roleProhibitions,
                _).asScala).toSet
53            val allRoles = plays.getRoles(player).diff(Seq(player))
54            val rs = if (allProhibitedRoles.size == allRoles.size) {
55              Set.empty[String]
56            } else {
57              allProhibitedRoles.filter(r => allRoles.exists(isInstanceOf(r, _)))
58            }
59            allProhibitedRoles.diff(rs).diff(list.toSet).foreach(r => if
                (allRoles.exists(isInstanceOf(r, _))) {
60              throw new RuntimeException(s"Role prohibition constraint violation:
                  '$player' plays role '$r', but it is not allowed to do so!")
61            })
62        }
63      }
64
65      /**
66        * Adds an role implication constraint between the given role types.
67        * Interpretation: if a core object plays an instance of role type A
68        * it also has to play an instance of role type B.
69        *
70        * @tparam A type of role A
71        * @tparam B type of role B that should be played implicitly if A is played
72        */
73      def RoleImplication[A : ClassTag, B : ClassTag](): Unit = {
74        val rA = classTag[A].toString
75        val rB = classTag[B].toString
76        val _ = roleImplications.putEdge(rA, rB)
77      }
78
79      /**
80        * Adds an role equivalent constraint between the given role types.
81        * Interpretation: if a core object plays an instance of role type A
82        * it also has to play an instance of role type B and visa versa.
83        *
84        * @tparam A type of role A that should be played implicitly if B is played
85        * @tparam B type of role B that should be played implicitly if A is played
86        */
87      def RoleEquivalence[A : ClassTag, B : ClassTag](): Unit = {
88        val rA = classTag[A].toString
89        val rB = classTag[B].toString
90        val _ = (roleEquivalents.putEdge(rA, rB), roleEquivalents.putEdge(rB, rA))
91      }
92
93      /**
94        * Adds an role prohibition constraint between the given role types.
95        * Interpretation: if a core object plays an instance of role type A
96        * it is not allowed to play B as well.
97        *
98        * @tparam A type of role A
99        * @tparam B type of role B that is not allowed to be played if A is
          played already
100       */
101     def RoleProhibition[A : ClassTag, B : ClassTag](): Unit = {
102       val rA = classTag[A].toString
103       val rB = classTag[B].toString
104       val _ = roleProhibitions.putEdge(rA, rB)
105     }
```

```
106
107     /**
108       * Wrapping function that checks all available role constraints for
109       * all core objects and its roles after the given function was executed.
110       * Throws a RuntimeException if a role constraint is violated!
111       *
112       * @param func the function to execute and check role constraints afterwards
113       */
114     def RoleConstraintsChecked(func: => Unit): Unit = {
115       func
116       plays.allPlayers.foreach(p => plays.getRoles(p).diff(Seq(p)).foreach(r =>
            validateConstraints(p, r)))
117     }
118
119     /**
120       * Checks all role constraints between the given player and role instance.
121       * Will throw a RuntimeException if a constraint is violated!
122       *
123       * @param player the player instance to check
124       * @param role   the role instance to check
125       */
126     private def validateConstraints(player: Any, role: Any): Unit = {
127       checkImplications(player, role)
128       checkEquivalence(player, role)
129       checkProhibitions(player, role)
130     }
131   }
```

D.2.6. ROLEGROUPS.SCALA

Listing D.11.: Source code for `RoleGroups.scala`.

```
package scroll.internal.support

import org.chocosolver.solver.{Model, Solution}
import org.chocosolver.solver.variables.IntVar
import scroll.internal.Compartment
import scroll.internal.util.ReflectiveHelper

import scala.reflect.{ClassTag, classTag}
import scala.collection.mutable

trait RoleGroups {
  self: Compartment =>

  private lazy val roleGroups = mutable.HashMap.empty[String, RoleGroup]

  private sealed trait Constraint

  private object AND extends Constraint

  private object OR extends Constraint

  private object XOR extends Constraint

  private object NOT extends Constraint

  /**
    * Wrapping function that checks all available role group constraints for
    * all core objects and its roles after the given function was executed.
    * Throws a RuntimeException if a role group constraint is violated!
    *
    * @param func the function to execute and check role group constraints afterwards
    */
  def RoleGroupsChecked(func: => Unit): Unit = {
    func
    validate()
  }

  private def validateOccurrenceCardinality(): Unit = {
    roleGroups.foreach { case (name, rg) =>
      val min = rg.occ._1
      val max = rg.occ._2
      val types = rg.getTypes
      val actual = types.map(ts => plays.allPlayers.count(r => ts == ReflectiveHelper.classSimpleClassName(r.getClass.toString))).sum
      if (actual < min  max < actual) {
        throw new RuntimeException(s"Occurrence cardinality in role group '$name' violated! " +
          s"Roles '$types' are played $actual times but should be between $min and $max.")
      }
    }
  }

  private def eval(rg: RoleGroup): Seq[String] = {
    val model = new Model("MODEL$" + rg.hashCode())
    val types = rg.getTypes
    val numOfTypes = types.size
```

```
55      val min = rg.limit._1
56      val max = rg.limit._2
57
58      val sumName = "SUM$" + rg.name
59      var sum = Option.empty[IntVar]
60      var op = Option.empty[Constraint]
61
62      // AND
63      if (max.compare(min) == 0 && min == numOfTypes) {
64        sum = Some(model.intVar(sumName, numOfTypes))
65        op = Some(AND)
66      }
67
68      // OR
69      if (min == 1 && max.compare(numOfTypes) == 0) {
70        sum = Some(model.intVar(sumName, 1, numOfTypes))
71        op = Some(OR)
72      }
73
74      // XOR
75      if (min == 1 && max.compare(1) == 0) {
76        sum = Some(model.intVar(sumName, 1))
77        op = Some(XOR)
78      }
79
80      // NOT
81      if (min == 0 && max.compare(0) == 0) {
82        sum = Some(model.intVar(sumName, 0))
83        op = Some(NOT)
84      }
85
86      val constrMap = types.map(ts => op match {
87        case Some(AND) => ts -> model.intVar("NUM$" + ts, 1)
88        case Some(OR) => ts -> model.intVar("NUM$" + ts, 0, numOfTypes)
89        case Some(XOR) => ts -> model.intVar("NUM$" + ts, 0, 1)
90        case Some(NOT) => ts -> model.intVar("NUM$" + ts, 0)
91        case None => throw new RuntimeException(s"Role group constraint of
           ($min, $max) for role group '${rg.name}' not possible!")
92      }).toMap
93
94      sum match {
95        case Some(s) =>
96          model.post(model.sum(constrMap.values.toArray, "=", s))
97        case None => throw new RuntimeException(s"Role group constraint of
           ($min, $max) for role group '${rg.name}' not possible!")
98      }
99
100     val solver = model.getSolver
101     if (solver.solve()) {
102       val resultRoleTypeSet = mutable.Set.empty[String]
103
104       val solutions = mutable.ListBuffer.empty[Solution]
105       do {
106         val sol = new Solution(model)
107         sol.record()
108         solutions += sol
109       } while (solver.solve())
110
111       val allPlayers = plays.allPlayers.filter(p =>
           !types.contains(ReflectiveHelper.classSimpleClassName(p.getClass.toString)))
112       if (allPlayers.forall(p => {
113         solutions.exists(s => {
```

```
114            types.forall(t => {
115              val numRole = plays.getRoles(p).count(r => t ==
                   ReflectiveHelper.classSimpleClassName(r.getClass.toString))
116              if (numRole == s.getIntVal(constrMap(t))) {
117                resultRoleTypeSet.add(t)
118                true
119              } else false
120            })
121          })
122        })) {
123          rg.evaluated = true
124          return resultRoleTypeSet.toSeq
125        }
126
127      } else {
128        throw new RuntimeException(s"Constraint set of role group '${rg.name}'
             unsolvable!")
129      }
130      // give up
131      throw new RuntimeException(s"Constraint set for inner cardinality of role
           group '${rg.name}' violated!")
132    }
133
134    private def validateInnerCardinality(): Unit = {
135      try {
136        roleGroups.values.filter(!_.evaluated).foreach(eval)
137      } finally {
138        roleGroups.values.foreach(_.evaluated = false)
139      }
140    }
141
142    /**
143      * Checks all role groups.
144      * Will throw a RuntimeException if a role group constraint is violated!
145      */
146    private def validate(): Unit = {
147      validateOccurrenceCardinality()
148      validateInnerCardinality()
149    }
150
151    private def addRoleGroup(rg: RoleGroup): RoleGroup = {
152      if (roleGroups.exists { case (n, _) => n == rg.name }) {
153        throw new RuntimeException(s"The RoleGroup ${rg.name} was already added!")
154      } else {
155        roleGroups(rg.name) = rg
156        rg
157      }
158    }
159
160    private type CInt = Ordered[Int]
161
162    trait Entry {
163      def getTypes: Seq[String]
164    }
165
166    object Types {
167      def apply(ts: String*): Types = new
           Types(ts.map(ReflectiveHelper.typeSimpleClassName))
168    }
169
170    class Types(ts: Seq[String]) extends Entry {
171      def getTypes: Seq[String] = ts
```

```
172   }
173
174   case class RoleGroup(name: String, entries: Seq[Entry], limit: (Int,
      ↪ CInt), occ: (Int, CInt), var evaluated: Boolean = false) extends
      ↪ Entry {
175     assert(0 <= occ._1 && occ._2 >= occ._1)
176     assert(0 <= limit._1 && limit._2 >= limit._1)
177
178     def getTypes: Seq[String] = entries.flatMap {
179       case ts: Types => ts.getTypes
180       case rg: RoleGroup => eval(rg)
181       case _ => throw new RuntimeException("Role groups can only contain a
          ↪ list of types or role groups itself!")
182     }
183   }
184
185   object RoleGroup {
186     private implicit def classTagToString(m: ClassTag[_]): String =
        ↪ ReflectiveHelper.simpleName(m.toString)
187
188     def apply(name: String) = new {
189
190       def containing(rg: RoleGroup*)(limit_l: Int, limit_u: CInt)(occ_l:
          ↪ Int, occ_u: CInt): RoleGroup =
191         addRoleGroup(new RoleGroup(name, rg, (limit_l, limit_u), (occ_l, occ_u)))
192
193       def containing[T1: ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int,
          ↪ occ_u: CInt): RoleGroup = {
194         val entry = Types(classTag[T1])
195         addRoleGroup(new RoleGroup(name, Seq(entry), (limit_l, limit_u),
            ↪ (occ_l, occ_u)))
196       }
197
198
199       def containing[T1: ClassTag, T2: ClassTag](limit_l: Int, limit_u:
          ↪ CInt)(occ_l: Int, occ_u: CInt): RoleGroup = {
200         val entry = Types(classTag[T1], classTag[T2])
201         addRoleGroup(new RoleGroup(name, Seq(entry), (limit_l, limit_u),
            ↪ (occ_l, occ_u)))
202       }
203
204       def containing[T1: ClassTag, T2: ClassTag, T3: ClassTag](limit_l:
          ↪ Int, limit_u: CInt)(occ_l: Int, occ_u: CInt): RoleGroup = {
205         val entry = Types(classTag[T1], classTag[T2], classTag[T3])
206         addRoleGroup(new RoleGroup(name, Seq(entry), (limit_l, limit_u),
            ↪ (occ_l, occ_u)))
207       }
208
209       def containing[T1: ClassTag, T2: ClassTag, T3: ClassTag, T4:
          ↪ ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int, occ_u: CInt):
          ↪ RoleGroup = {
210         val entry = Types(classTag[T1], classTag[T2], classTag[T3], classTag[T4])
211         addRoleGroup(new RoleGroup(name, Seq(entry), (limit_l, limit_u),
            ↪ (occ_l, occ_u)))
212       }
213
214
215       def containing[T1: ClassTag, T2: ClassTag, T3: ClassTag, T4:
          ↪ ClassTag, T5: ClassTag](limit_l: Int, limit_u: CInt)(occ_l: Int,
          ↪ occ_u: CInt): RoleGroup = {
216         val entry = Types(classTag[T1], classTag[T2], classTag[T3],
            ↪ classTag[T4], classTag[T5])
```

```
217           addRoleGroup(new RoleGroup(name, Seq(entry), (limit_l, limit_u),
              ↪   (occ_l, occ_u)))
218         }
219
220       }
221     }
222
223   }
```

D.2.7. ROLERESTRICTIONS.SCALA

Listing D.12.: Source code for RoleRestrictions.scala.

```
package scroll.internal.support

import java.lang.reflect.Method

import scroll.internal.util.ReflectiveHelper

import scala.collection.mutable
import scala.reflect.{ClassTag, classTag}

/**
  * Allows to add and check role restrictions (in the sense of structural
    typing) to a compartment instance.
  */
trait RoleRestrictions {
  private lazy val restrictions = mutable.HashMap.empty[String, List[Class[_]]]

  private def addToMap(m: mutable.Map[String, List[Class[_]]], elem:
      (String, List[Class[_]])): Unit = {
    val key = elem._1
    val value = elem._2
    if (m.contains(key)) {
      m(key) = m(key) ++ value
    } else {
      val _ = m += elem
    }
  }

  private def isInstanceOf(mani: String, that: String): Boolean =
    ReflectiveHelper.simpleName(that) == ReflectiveHelper.simpleName(mani)

  private def isSameInterface(roleInterface: Array[Method], restrInterface:
      Array[Method]): Boolean =
    restrInterface.forall(method => roleInterface.exists(method.equals))

  /**
    * Add a role restriction between the given player type A and role type B.
    *
    * @tparam A the player type
    * @tparam B the role type
    */
  def RoleRestriction[A: ClassTag, B: ClassTag](): Unit = {
    addToMap(restrictions, (classTag[A].toString,
      List(classTag[B].runtimeClass)))
  }

  /**
    * Replaces a role restriction for a player of type A with a
    * new role restriction between the given player type A and role type B.
    *
    * @tparam A the player type
    * @tparam B the role type
    */
  def ReplaceRoleRestriction[A: ClassTag, B: ClassTag](): Unit = {
    restrictions(classTag[A].toString) = List(classTag[B].runtimeClass)
  }

  /**
    * Checks all role restriction between the given player and a role type.
```

```
55     * Will throw a RuntimeException if a restriction is violated!
56     *
57     * @param player the player instance to check
58     * @param role   the role type to check
59     */
60   protected def validate[R : ClassTag](player: Any, role: R): Unit = {
61     val roleInterface = classTag[R].runtimeClass.getDeclaredMethods
62     restrictions.find { case (pt, rts) =>
63       isInstanceOf(pt, player.getClass.toString) && !rts.exists(r =>
           isSameInterface(roleInterface, r.getDeclaredMethods))
64     } match {
65       case Some((pt, rt)) => throw new RuntimeException(s"Role '$role' can not
           be played by '$player' due to the active role restrictions '$pt ->
           $rt'!")
66       case None => // fine, thanks
67     }
68   }
69 }
```

D.2.8. ROLEPLAYINGAUTOMATON.SCALA

Listing D.13.: Source code for RolePlayingAutomaton.scala.

```
package scroll.internal.rpa

import akka.actor._
import scroll.internal.Compartment
import scroll.internal.rpa.RolePlayingAutomaton.{RPAData, RPAState, Start,
  Stop, Uninitialized}

import scala.reflect.ClassTag

/**
  * Companion object for the [[scroll.internal.rpa.RolePlayingAutomaton]]
  containing
  * predefined states and data objects for messaging.
  */
object RolePlayingAutomaton {

  // some predefined states
  trait RPAState

  case object Start extends RPAState

  case object Stop extends RPAState

  // some predefined data objects for messaging
  trait RPAData

  case object Uninitialized extends RPAData

  case object BindRole extends RPAData

  case object RemoveRole extends RPAData

  case object TransferRole extends RPAData

  case object CheckConstraints extends RPAData

  case object Terminate extends RPAData

  def Use[T](implicit ct: ClassTag[T]) = new {
    def For(comp: Compartment): ActorRef =
      ActorSystem().actorOf(Props(ct.runtimeClass, comp), "rpa_" +
      comp.hashCode())
  }
}

/**
  * Use this trait to implement your own [[scroll.internal.Compartment]] specific
  * role playing automaton. Please read the documentation for [[akka.actor.FSM]]
  * carefully, since the features from that are applicable for role playing
  automatons.
  *
  * Remember to call <code>run()</code> when you want to start this
  automaton in your
  * [[scroll.internal.Compartment]] instance.
  *
  * This automaton will always start in state
  [[scroll.internal.rpa.RolePlayingAutomaton.Start]], so hook in there.
  *
```

```
52    * Final state is always [[scroll.internal.rpa.RolePlayingAutomaton.Stop]],
53    * that will terminate the actor system for this [[akka.actor.FSM]].
54    *
55    * Use the factory method <code>RolePlayingAutomaton.Use</code> to gain an instance of your specific FSM, e.g.:
56    *
57    * {{{
58    * trait MyCompartment extends Compartment {
59    * // ... some roles and interaction
60    *
61    * // your specific RPA here
62    * class MyRolePlayingAutomaton extends RolePlayingAutomaton {
63    * // specific behavior here
64    * when(Start) {
65    * // ...
66    * }
67    *
68    * onTransition {
69    * // ...
70    * }
71    *
72    * run()
73    * }
74    *
75    * Use[MyRolePlayingAutomaton] For this
76    * }
77    *
78    * // start everything
79    * new MyCompartment {}
80    * }}}
81    *
82    * Some predefined event types for messaging are available in the companion object.
83    * You may want to define your own states and event types.
84    * Simply use a companion object for this as well.
85    */
86  trait RolePlayingAutomaton extends Actor with LoggingFSM[RPAState, RPAData] {
87    /**
88      * Starts this automaton. Needs to be called first!
89      * Will set the initial state to [[scroll.internal.rpa.RolePlayingAutomaton.Start]].
90      */
91    def run(): Unit = {
92      log.debug(s"Starting RPA '${self.path}'")
93      startWith(Start, Uninitialized)
94      initialize()
95    }
96
97    /**
98      * Stops this automaton.
99      * Will set state to [[scroll.internal.rpa.RolePlayingAutomaton.Stop]] and terminates the
100     * actor system for this [[akka.actor.FSM]].
101     */
102   def halt(): State = {
103     context.system.terminate()
104     stop()
105   }
106
107   when(Stop) {
108     FSM.NullFunction
109   }
```

```
110
111    onTransition {
112      case _ -> Stop =>
113        log.debug(s"Stopping RPA '${self.path}'")
114        val _ = halt()
115    }
116  }
```

D.2.9. ROLEGRAPH.SCALA

Listing D.14.: Source code for RoleGraph.scala.

```
package scroll.internal.graph

import scroll.internal.support.DispatchQuery

import scala.reflect.ClassTag

/**
  * Trait defining an generic interface for all kind of role graphs.
  */
trait RoleGraph {
  /**
    * Merges this with another RoleGraph given as other.
    *
    * @param other the RoleGraph to merge with.
    */
  def merge(other: RoleGraph): Unit

  /**
    * Removes all players and plays-relationships specified in other from this RoleGraph.
    *
    * @param other the RoleGraph all players and plays-relationships specified in should removed from this
    */
  def detach(other: RoleGraph): Unit

  /**
    * Adds a plays relationship between core and role.
    *
    * @tparam P type of the player
    * @tparam R type of the role
    * @param player the player instance to add the given role
    * @param role   the role instance to add
    */
  def addBinding[P <: AnyRef : ClassTag, R <: AnyRef : ClassTag](player: P, role: R): Unit

  /**
    * Removes a plays relationship between core and role.
    *
    * @param player the player instance to remove the given role from
    * @param role   the role instance to remove
    */
  def removeBinding[P <: AnyRef : ClassTag, R <: AnyRef : ClassTag](player: P, role: R): Unit

  /**
    * Removes the given player from the graph.
    * This should remove its binding too!
    *
    * @param player the player to remove
    */
  def removePlayer[P <: AnyRef : ClassTag](player: P): Unit

  /**
    * Returns a Seq of all players
    *
    * @return a Seq of all players
```

```
55       */
56     def allPlayers: Seq[Any]
57
58     /**
59       * Returns a Seq of all roles attached to the given player (core object).
60       *
61       * @param player        the player instance to get the roles for
62       * @param dispatchQuery the strategy used to get all roles while
       ↪ traversing the role-playing graph, standard is DFS
63       * @return a Seq of all roles of core
64       */
65     def getRoles(player: Any)(implicit dispatchQuery: DispatchQuery =
         ↪ DispatchQuery.empty): Seq[Any]
66
67     /**
68       * Checks if the role graph contains the given player.
69       *
70       * @param player the player instance to check
71       * @return true if the role graph contains the given player, false otherwise
72       */
73     def containsPlayer(player: Any): Boolean
74
75     /**
76       * Returns a list of all predecessors of the given player, i.e. a
       ↪ transitive closure
77       * of its cores (deep roles).
78       *
79       * @param player        the player instance to calculate the cores of
80       * @param dispatchQuery the strategy used to get all predecessors while
       ↪ traversing the role-playing graph, standard is DFS
81       * @return a list of all predecessors of the given player
82       */
83     def getPredecessors(player: Any)(implicit dispatchQuery: DispatchQuery =
         ↪ DispatchQuery.empty): Seq[Any]
84   }
```

D.2.10. SCALAROLEGRAPH.SCALA

Listing D.15.: Source code for ScalaRoleGraph.scala.

```
package scroll.internal.graph

import com.google.common.graph.{GraphBuilder, Graphs}
import scroll.internal.support.DispatchQuery

import scala.reflect.ClassTag
import collection.JavaConverters._
import scala.collection.mutable

/**
  * Scala specific implementation of a [[scroll.internal.graph.RoleGraph]] using
  * a graph as underlying data model.
  *
  * @param checkForCycles set to true to forbid cyclic role playing relationships
  */
class ScalaRoleGraph(checkForCycles: Boolean = true) extends RoleGraph {

  private var root = GraphBuilder.directed().build[Object]()

  override def merge(other: RoleGraph): Unit = {
    require(null != other)
    require(other.isInstanceOf[ScalaRoleGraph], "You can only merge
      RoleGraphs of the same type!")

    val source = root
    val target = other.asInstanceOf[ScalaRoleGraph].root

    if (source.nodes().isEmpty && target.nodes().isEmpty) return

    if (source.nodes().isEmpty && !target.nodes().isEmpty) {
      root = target
      checkCycles()
      return
    }

    if (!source.nodes().isEmpty && target.nodes().isEmpty) return

    if (source.nodes().size < target.nodes().size) {
      source.edges().asScala.foreach(p => target.putEdge(p.source(), p.target()))
      root = target
    } else {
      target.edges().asScala.foreach(p => root.putEdge(p.source(), p.target()))
    }
    checkCycles()
  }

  override def detach(other: RoleGraph): Unit = {
    require(null != other)
    other.allPlayers.foreach(pl =>
      other.getRoles(pl).foreach(rl =>
        removeBinding(pl.asInstanceOf[AnyRef], rl.asInstanceOf[AnyRef])))
  }

  private def checkCycles(): Unit = {
    if (checkForCycles) {
      if (Graphs.hasCycle(root)) {
        throw new RuntimeException(s"Cyclic role-playing relationship found!")
      }
```

```
58      }
59    }
60
61    override def addBinding[P <: AnyRef : ClassTag, R <: AnyRef :
        ClassTag](player: P, role: R): Unit = {
62      require(null != player)
63      require(null != role)
64      root.putEdge(player, role)
65      if (checkForCycles && Graphs.hasCycle(root)) {
66        throw new RuntimeException(s"Cyclic role-playing relationship for player
          '$player' found!")
67      }
68    }
69
70    override def removeBinding[P <: AnyRef : ClassTag, R <: AnyRef :
        ClassTag](player: P, role: R): Unit = {
71      require(null != player)
72      require(null != role)
73      val _ = root.removeEdge(player, role)
74    }
75
76    override def removePlayer[P <: AnyRef : ClassTag](player: P): Unit = {
77      require(null != player)
78      val _ = root.removeNode(player)
79    }
80
81    override def getRoles(player: Any)(implicit dispatchQuery: DispatchQuery =
        DispatchQuery.empty): Seq[Any] = {
82      require(null != player)
83      Graphs.reachableNodes(root, player).asScala.toSeq
84    }
85
86    override def containsPlayer(player: Any): Boolean =
        root.nodes().contains(player)
87
88    override def allPlayers: Seq[Any] = root.nodes().asScala.toSeq
89
90    override def getPredecessors(player: Any)(implicit dispatchQuery:
        DispatchQuery = DispatchQuery.empty): Seq[Any] = {
91      val returnSeq = new mutable.ListBuffer[Any]
92      val processing = new mutable.Queue[Any]
93      root.predecessors(player).forEach(n => processing.enqueue(n))
94      while (processing.nonEmpty) {
95        val next = processing.dequeue()
96        if (!returnSeq.contains(next))
97          returnSeq += next
98        root.predecessors(next).forEach(n => processing.enqueue(n))
99      }
100     returnSeq
101   }
102 }
```

D.2.11. CACHEDSCALAROLEGRAPH.SCALA

Listing D.16.: Source code for CachedScalaRoleGraph.scala.

```
package scroll.internal.graph

import scroll.internal.support.DispatchQuery
import scroll.internal.util.Memoiser

import scala.reflect.ClassTag

class CachedScalaRoleGraph(checkForCycles: Boolean = true) extends
    ScalaRoleGraph(checkForCycles) with Memoiser {

  private class BooleanCache extends Memoised[Any, Boolean]

  private class SeqCache extends Memoised[Any, Seq[Any]]

  private val containsCache = new BooleanCache()
  private val predCache = new SeqCache()
  private val rolesCache = new SeqCache()

  override def addBinding[P <: AnyRef : ClassTag, R <: AnyRef :
      ClassTag](player: P, role: R): Unit = {
    super.addBinding(player, role)
    reset(player)
    reset(role)
  }

  private def resetAll(): Unit = {
    containsCache.reset()
    predCache.reset()
    rolesCache.reset()
  }

  private def reset(o: Any): Unit = {
    containsCache.resetAt(o)
    predCache.resetAt(o)
    rolesCache.resetAt(o)
  }

  override def containsPlayer(player: Any): Boolean =
    containsCache.getAndPutWithDefault(player, super.containsPlayer(player))

  override def detach(other: RoleGraph): Unit = {
    require(other.isInstanceOf[CachedScalaRoleGraph], "You can only detach
        RoleGraphs of the same type!")
    super.detach(other)
    resetAll()
  }

  override def getPredecessors(player: Any)(implicit dispatchQuery:
      DispatchQuery): Seq[Any] =
    predCache.getAndPutWithDefault(player, super.getPredecessors(player))

  override def getRoles(player: Any)(implicit dispatchQuery: DispatchQuery):
      Seq[Any] =
    rolesCache.getAndPutWithDefault(player, super.getRoles(player))

  override def merge(other: RoleGraph): Unit = {
    require(other.isInstanceOf[CachedScalaRoleGraph], "You can only merge
        RoleGraphs of the same type!")
```

```
53      super.merge(other)
54      resetAll()
55    }
56
57    override def removeBinding[P <: AnyRef : ClassTag, R <: AnyRef :
      ClassTag](player: P, role: R): Unit = {
58      super.removeBinding(player, role)
59      reset(player)
60      reset(role)
61    }
62
63    override def removePlayer[P <: AnyRef : ClassTag](player: P): Unit = {
64      super.removePlayer(player)
65      reset(player)
66    }
67  }
```

D.2.12. SCROLLERRORS.SCALA

Listing D.17.: Source code for SCROLLErrors.scala.

```
package scroll.internal.errors

/**
  * Object containing all SCROLL related error.
  */
object SCROLLErrors {

  sealed trait SCROLLError

  sealed trait TypeError

  sealed trait RolePlaying

  case class RolePlayingImpossible(core: String, role: String) extends RolePlaying

  case class TypeNotFound(name: String) extends TypeError

  case class RoleNotFound(forCore: String, target: String, args: String) extends SCROLLError

  sealed trait InvocationError extends SCROLLError

  case class IllegalRoleInvocationSingleDispatch(roleType: String, target: String) extends InvocationError

  case class IllegalRoleInvocationMultipleDispatch(roleType: String, target: String, args: String) extends InvocationError

}
```

D.2.13. MEMOISER.SCALA

Listing D.18.: Source code for Memoiser.scala.

```
package scroll.internal.util

import com.google.common.cache.{Cache, CacheBuilder}

/**
  * Support for memoization, encapsulating common behaviour of memoised
  * entities and a general reset mechanism for all such entities.
  */
trait Memoiser {

  /**
    * Common interface for encapsulation of memoization for a single memoised
    * entity backed by a configurable cache.
    */
  trait MemoisedBase[T, U] {

    /**
      * The memo table.
      */
    def memo: Cache[AnyRef, AnyRef]

    /**
      * Return the value stored at key `t` as an option.
      */
    def get(t: T): Option[U] = Option(memo.getIfPresent(t).asInstanceOf[U])

    /**
      * Return the value stored at key `t` if there is one, otherwise
      * return `u`. `u` is only evaluated if necessary and put into the cache.
      */
    def getAndPutWithDefault(t: T, u: => U): U = get(t) match {
      case Some(v) => v
      case None =>
        val newU = u
        put(t, newU)
        newU
    }

    /**
      * Has the value at `t` already been computed or not? By default, does
      * the memo table contain a value for `t`?
      */
    def hasBeenComputedAt(t: T): Boolean = get(t).isDefined

    /**
      * Store the value `u` under the key `t`.
      */
    def put(t: T, u: U): Unit = {
      memo.put(t.asInstanceOf[AnyRef], u.asInstanceOf[AnyRef])
    }

    /**
      * Immediately reset the memo table.
      */
    def reset(): Unit = {
      memo.invalidateAll()
    }

```

```
59      /**
60        * Immediately reset the memo table at `t`.
61        */
62      def resetAt(t: T): Unit = {
63        memo.invalidate(t)
64      }
65
66      /**
67        * The number of entries in the memo table.
68        */
69      def size(): Long = memo.size
70    }
71
72    /**
73      * A memoised entity that uses equality to compare keys.
74      */
75    trait Memoised[T, U] extends MemoisedBase[T, U] {
76      val memo: Cache[AnyRef, AnyRef] = CacheBuilder.newBuilder.build()
77    }
78
79    /**
80      * A memoised entity that weakly holds onto its keys and uses identity
81      * to compare them.
82      */
83    trait IdMemoised[T, U] extends MemoisedBase[T, U] {
84      val memo: Cache[AnyRef, AnyRef] = CacheBuilder.newBuilder.weakKeys.build()
85    }
86
87  }
```

D.2.14. REFLECTIVEHELPER.SCALA

Listing D.19.: Source code for ReflectiveHelper.scala.

```
1   package scroll.internal.util
2
3   import scala.annotation.tailrec
4   import scala.reflect.{ClassTag, classTag}
5
6   /**
7     * Contains useful functions for translating class and type names to Strings
8     * and provides helper functions to access common tasks for working with
      reflections.
9     *
10    * Querying methods and fields is cached using
      [[scroll.internal.util.Memoiser]].
11    */
12  object ReflectiveHelper extends Memoiser {
13
14    import java.lang
15    import java.lang.reflect.{Field, Method}
16
17    private class MethodCache extends Memoised[Any, Set[Method]]
18
19    private class FieldCache extends Memoised[Any, Set[Field]]
20
21    private class SimpleTagNameCache extends Memoised[ClassTag[_], String]
22
23    private class SimpleClassNameCache extends Memoised[Class[_], String]
24
25    private lazy val methodCache = new MethodCache()
26    private lazy val fieldCache = new FieldCache()
27    private lazy val simpleClassNameCache = new SimpleClassNameCache()
28    private lazy val simpleTagNameCache = new SimpleTagNameCache()
29
30    private def simpleClassName(s: String, on: String) = if (s.contains(on)) {
31      s.substring(s.lastIndexOf(on) + 1)
32    } else {
33      s
34    }
35
36    /**
37      * Translates a Type name to a String, i.e. removing anything before the last
38      * occurrence of "<code>.</code>".
39      *
40      * @param t the Type name as String
41      * @return anything after the last occurrence of "<code>.</code>"
42      */
43    def typeSimpleClassName(t: String): String = simpleClassName(t, ".")
44
45    /**
46      * Translates a Class name to a String, i.e. removing anything before the last
47      * occurrence of "<code>$</code>".
48      *
49      * @param t the Class name as String
50      * @return anything after the last occurrence of "<code>$</code>"
51      */
52    def classSimpleClassName(t: String): String = simpleClassName(t, "$")
53
54    /**
55      * Translates a Class or Type name to a String, i.e. removing anything
      before the last
```

```
56     * occurrence of "<code>$</code>" or "<code>.</code>".
57     *
58     * @param t the Class or Type name as String
59     * @return anything after the last occurrence of "<code>$</code>" or
       "<code>.</code>"
60     */
61   def simpleName(t: String): String =
       typeSimpleClassName(classSimpleClassName(t))
62
63   /**
64     * Returns the hash code of any object as String.
65     *
66     * @param of the object to get the hash code as String
67     * @return the hash code of 'of' as String.
68     */
69   def hash(of: Any): String = of.hashCode().toString
70
71
72   private def safeString(s: String): Unit = {
73     require(null != s)
74     require(!s.isEmpty)
75   }
76
77   @tailrec
78   private def safeFindField(of: Any, name: String): Field =
       fieldCache.get(of) match {
79     case Some(fields) => fields.find(_.getName == name) match {
80       case Some(f) => f
81       case None => throw new RuntimeException(s"Field '$name' not found on
           '$of'!")
82     }
83     case None =>
84       val fields = getAllFields(of)
85       fieldCache.put(of, fields)
86       safeFindField(of, name)
87   }
88
89   @tailrec
90   private def findMethods(of: Any, name: String): Set[Method] =
       methodCache.get(of) match {
91     case Some(l) =>
92       l.filter(_.getName == name)
93     case None =>
94       val methods = getAllMethods(of)
95       methodCache.put(of, methods)
96       findMethods(of, name)
97   }
98
99   private def getAllMethods(of: Any): Set[Method] = {
100    def getAccessibleMethods(c: Class[_]): Set[Method] = c match {
101      case null => Set.empty
102      case _ => c.getDeclaredMethods.toSet ++
           getAccessibleMethods(c.getSuperclass)
103    }
104
105    getAccessibleMethods(of.getClass)
106  }
107
108  private def getAllFields(of: Any): Set[Field] = {
109    def getAccessibleFields(c: Class[_]): Set[Field] = c match {
110      case null => Set.empty
111      case _ => c.getDeclaredFields.toSet ++ getAccessibleFields(c.getSuperclass)
```

```
112      }
113
114      getAccessibleFields(of.getClass)
115    }
116
117    private def matchMethod[A](m: Method, name: String, args: Seq[A]): Boolean = {
118      lazy val matchName = m.getName == name
119      lazy val matchParamCount = m.getParameterTypes.length == args.size
120      lazy val matchArgTypes = args.zip(m.getParameterTypes).forall {
121        case (arg, paramType: Class[_]) => paramType match {
122          case lang.Boolean.TYPE => arg.isInstanceOf[Boolean]
123          case lang.Character.TYPE => arg.isInstanceOf[Char]
124          case lang.Short.TYPE => arg.isInstanceOf[Short]
125          case lang.Integer.TYPE => arg.isInstanceOf[Integer]
126          case lang.Long.TYPE => arg.isInstanceOf[Long]
127          case lang.Float.TYPE => arg.isInstanceOf[Float]
128          case lang.Double.TYPE => arg.isInstanceOf[Double]
129          case lang.Byte.TYPE => arg.isInstanceOf[Byte]
130          case _ => paramType.isAssignableFrom(arg.getClass)
131        }
132        case faultyArgs => throw new RuntimeException(s"Can not handle this
           arguments: '$faultyArgs'")
133      }
134      matchName && matchParamCount && matchArgTypes
135    }
136
137    /**
138      * @return all methods/functions of the wrapped object as Set
139      */
140    def allMethods(of: Any): Set[Method] = methodCache.get(of) match {
141      case Some(methods) => methods
142      case None =>
143        val methods = getAllMethods(of)
144        methodCache.put(of, methods)
145        methods
146    }
147
148    /**
149      * Find a method of the wrapped object by its name and argument list given.
150      *
151      * @param on   the instance to search on
152      * @param name the name of the function/method of interest
153      * @param args the args function/method of interest
154      * @return Some(Method) if the wrapped object provides the
         function/method in question, None otherwise
155      */
156    def findMethod(on: Any, name: String, args: Seq[Any]): Option[Method] =
        findMethods(on, name).find(matchMethod(_, name, args))
157
158    /**
159      * Checks if the wrapped object provides a member (field or
         function/method) with the given name.
160      *
161      * @param on   the instance to search on
162      * @param name the name of the member (field or function/method)  of interest
163      * @return true if the wrapped object provides the given member, false
         otherwise
164      */
165    def hasMember(on: Any, name: String): Boolean = {
166      safeString(name)
167
168      val fields = fieldCache.get(on) match {
```

```
169        case Some(fs) => fs
170        case None =>
171          val fs = getAllFields(on)
172          fieldCache.put(on, fs)
173          fs
174      }
175
176      val methods = methodCache.get(on) match {
177        case Some(ms) => ms
178        case None =>
179          val ms = getAllMethods(on)
180          methodCache.put(on, ms)
181          ms
182      }
183
184      fields.exists(_.getName == name)  methods.exists(_.getName == name)
185    }
186
187    /**
188      * Returns the runtime content of type T of the field with the given name
         of the wrapped object.
189      *
190      * @param on   the instance to search on
191      * @param name the name of the field of interest
192      * @tparam T the type of the field
193      * @return the runtime content of type T of the field with the given name
         of the wrapped object
194      */
195    def propertyOf[T](on: Any, name: String): T = {
196      safeString(name)
197      val field = safeFindField(on, name)
198      field.setAccessible(true)
199      field.get(on).asInstanceOf[T]
200    }
201
202    /**
203      * Sets the field given as name to the provided value.
204      *
205      * @param on    the instance to search on
206      * @param name  the name of the field of interest
207      * @param value the value to set for this field
208      */
209    def setPropertyOf(on: Any, name: String, value: Any): Unit = {
210      safeString(name)
211      val field = safeFindField(on, name)
212      field.setAccessible(true)
213      field.set(on, value)
214    }
215
216    /**
217      * Returns the runtime result of type T of the given function by executing
         this function of the wrapped object.
218      *
219      * @param on the instance to search on
220      * @param m  the function of interest
221      * @tparam T the return type of the function
222      * @return the runtime result of type T of the function with the given
         name by executing this function of the wrapped object
223      */
224    def resultOf[T](on: Any, m: Method): T = {
225      m.setAccessible(true)
226      m.invoke(on).asInstanceOf[T]
```

```
227   }
228
229   /**
230    * Returns the runtime result of type T of the given function and
       arguments by executing this function of the wrapped object.
231    *
232    * @param on   the instance to search on
233    * @param m    the function of interest
234    * @param args the arguments of the function of interest
235    * @tparam T the return type of the function
236    * @return the runtime result of type T of the function with the given
       name by executing this function of the wrapped object
237    */
238   def resultOf[T](on: Any, m: Method, args: Seq[Object]): T = {
239     m.setAccessible(true)
240     m.invoke(on, args: _*).asInstanceOf[T]
241   }
242
243   /**
244    * Returns the runtime result of type T of the function with the given
       name by executing this function of the wrapped object.
245    *
246    * @param on   the instance to search on
247    * @param name the name of the function of interest
248    * @tparam T the return type of the function
249    * @return the runtime result of type T of the function with the given
       name by executing this function of the wrapped object
250    */
251   def resultOf[T](on: Any, name: String): T = {
252     safeString(name)
253     findMethods(on, name).toList match {
254       case elem :: Nil =>
255         elem.setAccessible(true)
256         elem.invoke(on).asInstanceOf[T]
257       case list if list.nonEmpty =>
258         val elem = list.head
259         elem.setAccessible(true)
260         elem.invoke(on).asInstanceOf[T]
261       case Nil =>
262         throw new RuntimeException(s"Function with name '$name' not found on
             '$on'!")
263     }
264   }
265
266   /**
267    * Checks if the wrapped object is of type T.
268    *
269    * @param on the instance to search on
270    * @tparam T the type to check
271    * @return true if the wrapped object is of type T, false otherwise
272    */
273   def is[T: ClassTag](on: Any): Boolean =
274     simpleClassNameCache.getAndPutWithDefault(on.getClass,
        ReflectiveHelper.simpleName(on.getClass.toString)) ==
275       simpleTagNameCache.getAndPutWithDefault(classTag[T],
          ReflectiveHelper.simpleName(classTag[T].toString))
276 }
```

D.3. SOURCE CODE FOR EVALUATION

D.3.1. ROP

D.3.1.1. ROP/BANKEXAMPLE.SCALA

Listing D.20.: Source code for rop/BankExample.scala.

```
1   package rop
2
3   import common.{Benchmarkable, Currency => Money}
4   import java.util.{Date => DateTime}
5
6   import scala.collection.mutable
7   import scala.util.Random
8
9   class BankExample extends Benchmarkable {
10
11    case class Person(title: String, firstName: String, lastName: String,
        address: String) extends ComponentCore
12
13    case class Account(var balance: Money, id: Integer) extends ComponentCore {
14      def increase(amount: Money): Unit = {
15        balance = balance + amount
16      }
17
18      def decrease(amount: Money): Unit = {
19        balance = balance - amount
20      }
21    }
22
23    class Transaction(amount: Money, creationTime: DateTime, from: Account, to:
        Account) extends ComponentCore {
24      def execute(): Boolean = {
25        Source().withdraw(amount)
26        Target().deposite(amount)
27        true
28      }
29
30      case class Source() extends ComponentRole(from) {
31        def withdraw(amount: Money): Unit = {
32          val _ = core.asInstanceOf[ComponentCore].getRoles.collectFirst {
33            case ca: Bank#CheckingsAccount => ca.decrease(amount)
34            case sa: Bank#SavingsAccount => sa.decrease(amount)
35          }
36        }
37      }
38
39      case class Target() extends ComponentRole(to) {
40        def deposite(amount: Money): Unit = {
41          core.asInstanceOf[Account].increase(amount)
42        }
43      }
44
45    }
46
47    trait Bank {
48      var name: String = _
49
50      var moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]
51
52      def executeTransactions(): Unit = {
```

```
53        moneyTransfers.foreach(_.execute())
54      }
55
56      case class Customer(name: String, id: Integer, p: Person) extends
          ComponentRole(p) {
57        var accounts = mutable.ArrayBuffer.empty[Account]
58
59        def addSavingsAccount(a: Account): Boolean = {
60          val sa = SavingsAccount(0.1, a)
61          a.addRole(sa)
62          accounts.append(a)
63          true
64        }
65
66        def addCheckingsAccount(a: Account): Boolean = {
67          val ca = CheckingsAccount(Money(100, "USD"), a)
68          a.addRole(ca)
69          accounts.append(a)
70          true
71        }
72      }
73
74      case class MoneyTransfer(execution: DateTime, t: Transaction) extends
          ComponentRole(t) {
75        var executed: Boolean = false
76
77        def execute(): Boolean = {
78          core.asInstanceOf[Transaction].execute()
79          executed = true
80          isExecuted
81        }
82
83        def isExecuted: Boolean = executed
84      }
85
86      case class CheckingsAccount(limit: Money, a: Account) extends
          ComponentRole(a) {
87        def decrease(amount: Money): Unit = amount match {
88          case am if am <= limit =>
89            val _ = core.asInstanceOf[Account].decrease(amount)
90          case _ => throw new IllegalArgumentException("Amount > limit!")
91        }
92      }
93
94      case class SavingsAccount(var transactionFee: Double, a: Account) extends
          ComponentRole(a) {
95        def decrease(amount: Money): Unit = {
96          val _ = core.asInstanceOf[Account].decrease(amount + amount *
              transactionFee)
97        }
98      }
99
100   }
101
102   var bank: Bank = _
103
104   override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
        checkCycles: Boolean = false): BankExample = {
105     val players = (0 until numPlayer).map(i => Person("Mr.", "Stan", "Mejer" +
          i, "Fake Street 1A"))
106
107     bank = new Bank {
```

```
108        name = "Deutsche Bank"
109
110        val accounts = players.zipWithIndex.map { case (p, i) =>
111          val a = Account(Money(100.0, "USD"), i)
112          (0 until numRoles).map(ii => {
113            Customer("Customer", ii, p).addSavingsAccount(a)
114          })
115          a
116        }
117
118        (0 until numTransactions).foreach { _ =>
119          val s = accounts(Random.nextInt(accounts.size))
120          val t = accounts(Random.nextInt(accounts.size))
121          val transaction = new Transaction(Money(10.0, "USD"), new DateTime, s, t)
122          val mt = MoneyTransfer(new DateTime, transaction)
123          moneyTransfers.append(mt)
124        }
125      }
126      this
127    }
128
129    override def benchmark(): Unit = {
130      bank.executeTransactions()
131    }
132  }
```

D.3.1.2. ROP/COMPONENT.SCALA

Listing D.21.: Source code for rop/Component.scala.

```
1   package rop
2
3   abstract class Component {
4
5     def addRole(spec: ComponentRole): Unit
6
7     def getRole(spec: String): Iterable[ComponentRole]
8
9     def hasRole(spec: String): Boolean
10
11    def as(role: ComponentRole): ComponentRole = {
12      role.setCore(this)
13      role
14    }
15  }
```

D.3.1.3. ROP/COMPONENTCORE.SCALA

Listing D.22.: Source code for rop/ComponentCore.scala.

```
1   package rop
2
3   import collection.mutable.ArrayBuffer
4   import scala.collection.mutable
5
6   class ComponentCore extends Component {
7     private val roles = mutable.Map[String, ArrayBuffer[ComponentRole]]()
8
9     override def addRole(spec: ComponentRole): Unit = {
10      if (roles.contains(spec.getName)) {
11        roles(spec.getName) = roles(spec.getName) :+ spec
12      } else {
```

```
13        roles(spec.getName) = ArrayBuffer(spec)
14      }
15    }
16
17    override def getRole(spec: String): Iterable[ComponentRole] = roles(spec)
18
19    override def hasRole(spec: String): Boolean = roles.contains(spec)
20
21    def getRoles: Iterable[ComponentRole] = roles.values.flatten
22  }
```

D.3.1.4. ROP/COMPONENTROLE.SCALA

Listing D.23.: Source code for rop/ComponentRole.scala.

```
1   package rop
2
3   class ComponentRole(var core: Component) extends Component {
4     def getName = this.getClass.getSimpleName
5
6     def setCore(core: Component): Unit = {
7       this.core = core
8     }
9
10    override def addRole(spec: ComponentRole): Unit = {
11      core.addRole(spec)
12    }
13
14    override def getRole(spec: String): Iterable[ComponentRole] = {
15      core.getRole(spec)
16    }
17
18    override def hasRole(spec: String): Boolean = {
19      core.hasRole(spec)
20    }
21  }
```

D.3.2. SCALAROLES

D.3.2.1. SCALAROLES/BANKEXAMPLE.SCALA

Listing D.24.: Source code for `rop/scalaroles/BankExample.scala`.

```
package scalaroles

import java.util.{Date => DateTime}

import common.{Benchmarkable, Currency => Money}

import scala.collection.mutable
import scala.util.Random

class BankExample extends Benchmarkable {

  case class Person(title: String, firstName: String, lastName: String,
    address: String)

  case class Account(var balance: Money, id: Integer) {
    def increase(amount: Money): Unit = {
      balance = balance + amount
    }

    def decrease(amount: Money): Unit = {
      balance = balance - amount
    }
  }

  trait Transaction extends StickyCollaboration {
    var amount: Money = _
    var creationTime: DateTime = _

    val source = new Source {}
    val target = new Target {}

    // TODO: deep roles are a big problem here. How to delegate to Savings-
      or CheckingsAccount?

    trait Source extends Role[Account] {
      def withdraw(): Unit = {
        core.decrease(amount)
      }
    }

    trait Target extends Role[Account] {
      def deposite(): Unit = {
        core.increase(amount)
      }
    }

    def execute(): Boolean = {
      source.withdraw()
      target.deposite()
      true
    }
  }

  trait Bank extends StickyCollaboration {

    var name: String = _

```

```
56      var moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]
57
58      def executeTransactions(): Unit = {
59        moneyTransfers.foreach(_.execute())
60      }
61
62      trait Customer extends Role[Person] {
63        var name: String = _
64        var id: Integer = _
65
66        var accounts = mutable.ArrayBuffer.empty[Account]
67
68        def addSavingsAccount(a: Account): Boolean = {
69          val sa = new SavingsAccount {
70            transactionFee = 0.1
71          }
72          accounts.append(a)
73          sa.playedBy(a)
74          true
75        }
76
77        def addCheckingsAccount(a: Account): Boolean = {
78          val ca = new CheckingsAccount {
79            limit = Money(100, "USD")
80          }
81          accounts.append(a)
82          ca.playedBy(a)
83          true
84        }
85
86      }
87
88      trait MoneyTransfer extends Role[Transaction] {
89        var execution: DateTime = _
90        var executed: Boolean = false
91
92        def execute(): Boolean = {
93          core.execute()
94          executed = true
95          isExecuted
96        }
97
98        def isExecuted: Boolean = executed
99      }
100
101     trait CheckingsAccount extends Role[Account] {
102       var limit: Money = _
103
104       def decrease(amount: Money): Unit = amount match {
105         case a if a <= limit => core.decrease(amount)
106         case _ => throw new IllegalArgumentException("Amount > limit!")
107       }
108     }
109
110     trait SavingsAccount extends Role[Account] {
111       var transactionFee: Double = _
112
113       def decrease(amount: Money): Unit = {
114         core.decrease(amount + amount * transactionFee)
115       }
116     }
117
```

```
118     }
119 
120     var bank: Bank = _
121 
122     override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
         ↪ checkCycles: Boolean = false): BankExample = {
123       val players = (0 until numPlayer).map(i => Person("Mr.", "Stan", "Mejer" +
           ↪ i, "Fake Street 1A"))
124 
125       bank = new Bank {
126         name = "Deutsche Bank"
127 
128         val accounts = players.zipWithIndex.map { case (p, i) =>
129           val a = Account(Money(100.0, "USD"), i)
130           (0 until numRoles).map(ii => {
131             val c = new Customer {
132               name = "Customer"
133               id = ii
134             }
135             c.playedBy(p)
136             c.addSavingsAccount(a)
137           })
138           a
139         }
140 
141         (0 until numTransactions).foreach { _ =>
142           val transaction = new Transaction {
143             amount = Money(10.0, "USD")
144             creationTime = new DateTime
145             source.playedBy(accounts(Random.nextInt(accounts.size)))
146             target.playedBy(accounts(Random.nextInt(accounts.size)))
147           }
148 
149           val transfer = new MoneyTransfer {
150             execution = new DateTime()
151           }
152           transfer.playedBy(transaction)
153           moneyTransfers.append(transfer)
154         }
155       }
156       this
157     }
158 
159     override def benchmark(): Unit = {
160       bank.executeTransactions()
161     }
162   }
```

D.3.3. SCROLL

D.3.3.1. SCROLL/BANKEXAMPLE.SCALA

Listing D.25.: Source code for SCROLL/BankExample.scala.

```
package scroll

import scroll.internal.support.DispatchQuery
import DispatchQuery._
import scroll.internal.Compartment
import common.{Benchmarkable, Currency => Money}

import scroll.internal.graph.CachedScalaRoleGraph

import scala.collection.mutable
import scala.util.Random

class BankExample extends Benchmarkable {

  case class Person(title: String, firstName: String, lastName: String,
      address: String)

  case class Account(var balance: Money, id: Integer) {

    def increase(amount: Money): Unit = {
      balance = balance + amount
    }

    def decrease(amount: Money): Unit = {
      balance = balance - amount
    }
  }

  trait Transaction extends Compartment {
    var amount: Money = _

    var from: Source = _
    var to: Target = _

    def execute(): Boolean = {
      from.withdraw(amount)
      to.deposite(amount)
      true
    }

    case class Source() {
      def withdraw(amount: Money): Unit = {
        val _ = +this decrease amount
      }
    }

    case class Target() {
      def deposite(amount: Money): Unit = {
        val _ = +this increase amount
      }
    }

  }

  trait Bank extends Compartment {
    var moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]

```

```
57      def executeTransactions(): Unit = {
58        moneyTransfers.foreach(_.execute())
59      }
60
61      case class Customer(name: String, id: Integer) {
62        var accounts = mutable.ArrayBuffer.empty[Account]
63
64        def addSavingsAccount(a: Account): Boolean = {
65          val sa = SavingsAccount(0.1)
66          accounts.append(a)
67          a play sa
68          true
69        }
70
71        def addCheckingsAccount(a: Account): Boolean = {
72          val ca = CheckingsAccount(Money(100, "USD"))
73          accounts.append(a)
74          a play ca
75          true
76        }
77      }
78
79      case class MoneyTransfer() {
80        def execute(): Boolean = {
81          implicit val dd = Bypassing(_.isInstanceOf[MoneyTransfer])
82          +this execute()
83          true
84        }
85      }
86
87      case class CheckingsAccount(var limit: Money) {
88        def decrease(amount: Money): Unit = amount match {
89          case a if a <= limit =>
90            implicit val dd = Bypassing(_.isInstanceOf[CheckingsAccount])
91            val _ = +this decrease amount
92          case _ => throw new IllegalArgumentException("Amount > limit!")
93        }
94      }
95
96      case class SavingsAccount(var transactionFee: Double) {
97        def decrease(amount: Money): Unit = {
98          implicit val dd = Bypassing(_.isInstanceOf[SavingsAccount])
99          //println("dec from SA")
100         val _ = +this decrease (amount + amount * transactionFee)
101       }
102     }
103
104   }
105
106   var bank: Bank = _
107
108   override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
        checkCycles: Boolean = false): BankExample = {
109     val players = (0 until numPlayer).map(i => Person("Mr.", "Stan", "Mejer" +
          i, "Fake Street 1A"))
110
111     bank = new Bank {
112       override val plays = new CachedScalaRoleGraph(checkCycles)
113
114       val accounts = players.zipWithIndex.map { case (p, i) =>
115         val a = Account(Money(100.0, "USD"), i)
116         (0 until numRoles).map(ii => {
```

```
117           val c = Customer("Customer", ii)
118           p play c
119           c addSavingsAccount a
120         })
121         a
122       }
123
124       (0 until numTransactions).foreach { _ =>
125         val transaction = new Transaction {
126           override val plays = new CachedScalaRoleGraph(checkCycles)
127           amount = Money(10.0, "USD")
128           from = Source()
129           to = Target()
130           accounts(Random.nextInt(accounts.size)) play from
131           accounts(Random.nextInt(accounts.size)) play to
132         }
133         val mt = MoneyTransfer()
134         transaction play mt
135         moneyTransfers.append(mt)
136         transaction partOf this
137       }
138     }
139     this
140   }
141
142   override def benchmark(): Unit = {
143     bank.executeTransactions()
144   }
145
146 }
```

D.3.4. SEPARATETYPE

D.3.4.1. SEPARATETYPE/BANKEXAMPLE.SCALA

Listing D.26.: Source code for SeparateType/BankExample.scala.

```
package separatetype

import common.{Benchmarkable, Currency => Money}
import java.util.{Date => DateTime}

import scala.collection.mutable
import scala.util.Random

class BankExample extends Benchmarkable {

  case class Person(title: String, firstName: String, lastName: String,
      address: String)

  case class Account(var balance: Money, id: Integer, accountType: String) {
    def increase(amount: Money): Unit = {
      balance = balance + amount
    }

    def decrease(amount: Money): Unit = {
      balance = balance - amount
    }
  }

  case class Source(a: Account) {
    def withdraw(amount: Money): Unit = a.accountType match {
      case "Savings" => SavingsAccount(0.1, a).decrease(amount)
      case "Checkings" => CheckingsAccount(Money(100.0, "USD"),
          a).decrease(amount)
    }
  }

  case class Target(a: Account) {
    def deposite(amount: Money): Unit = {
      val _ = a increase amount
    }
  }

  case class Customer(name: String, id: Integer, p: Person) {
    var accounts = mutable.ArrayBuffer.empty[Account]

    def addSavingsAccount(a: Account): Boolean = {
      accounts.append(a)
      true
    }

    def addCheckingsAccount(a: Account): Boolean = {
      accounts.append(a)
      true
    }
  }

  case class Transaction(amount: Money, creationTime: DateTime, from: Source,
      to: Target) {
    def execute(): Boolean = {
      from.withdraw(amount)
      to.deposite(amount)
      true
```

```
55        }
56      }
57
58      case class MoneyTransfer(execution: DateTime, t: Transaction) {
59        var executed: Boolean = false
60
61        def execute(): Boolean = {
62          t.execute()
63          executed = true
64          isExecuted
65        }
66
67        def isExecuted: Boolean = executed
68      }
69
70      case class CheckingsAccount(var limit: Money, acc: Account) {
71        def decrease(amount: Money): Unit = amount match {
72          case a if a <= limit =>
73            val _ = acc decrease amount
74          case _ => throw new IllegalArgumentException("Amount > limit!")
75        }
76      }
77
78      case class SavingsAccount(var transactionFee: Double, acc: Account) {
79        def decrease(amount: Money): Unit = {
80          val _ = acc decrease (amount + amount * transactionFee)
81        }
82      }
83
84      trait Bank {
85        var name: String = _
86
87        var moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]
88
89        def executeTransactions(): Unit = {
90          moneyTransfers.foreach(_.execute())
91        }
92      }
93
94      var bank: Bank = _
95
96      override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
          checkCycles: Boolean = false): BankExample = {
97        val players = (0 until numPlayer).map(i => Person("Mr.", "Stan", "Mejer" +
            i, "Fake Street 1A"))
98
99        bank = new Bank {
100         name = "Deutsche Bank"
101
102         val accounts = players.zipWithIndex.map { case (p, i) =>
103           val a = Account(Money(100.0, "USD"), i, "Savings")
104           (0 until numRoles).map(ii => {
105             val c = Customer(p.lastName, i, p)
106             c.addSavingsAccount(a)
107           })
108           a
109         }
110
111         (0 until numTransactions).foreach { _ =>
112           val source = Source(accounts(Random.nextInt(accounts.size)))
113           val target = Target(accounts(Random.nextInt(accounts.size)))
```

```
114           val transaction = Transaction(Money(10.0, "USD"), new DateTime,
                  source, target)
115           val mt = MoneyTransfer(new DateTime, transaction)
116           moneyTransfers.append(mt)
117         }
118       }
119       this
120     }
121 
122     override def benchmark(): Unit = {
123       bank.executeTransactions()
124     }
125 
126   }
```

D.3.5. SINGLETYPE

D.3.5.1. SINGLETYPE/BANKEXAMPLE.SCALA

Listing D.27.: Source code for SingleType/BankExample.scala.

```
package singletype

import common.{Benchmarkable, Currency => Money}
import java.util.{Date => DateTime}

import scala.collection.mutable
import scala.util.Random

class BankExample extends Benchmarkable {

  case class Person(title: String, firstName: String, lastName: String,
      address: String) {
    var accounts = mutable.ArrayBuffer.empty[Account]

    def addSavingsAccount(a: Account): Boolean = {
      accounts.append(a)
      true
    }

    def addCheckingsAccount(a: Account): Boolean = {
      accounts.append(a)
      true
    }
  }

  case class Account(var balance: Money, id: Integer, fee: Double, limit:
      Money, accountType: String) {
    def increase(amount: Money): Unit = {
      balance = balance + amount
    }

    def decrease(amount: Money): Unit = {
      balance = balance - amount
    }

    def withdraw(amount: Money): Unit = accountType match {
      case "Savings" => decreaseWithFee(amount)
      case "Checkings" => decreaseWithLimit(amount)
    }

    def deposite(amount: Money): Unit = {
      val _ = this increase amount
    }

    def decreaseWithLimit(amount: Money): Unit = amount match {
      case a if a <= limit =>
        val _ = this decrease amount
      case _ => throw new IllegalArgumentException("Amount > limit!")
    }

    def decreaseWithFee(amount: Money): Unit = {
      val _ = this decrease (amount + amount * fee)
    }
  }

  case class Transaction(amount: Money, source: Account, target: Account,
      creationTime: DateTime) {
```

```
55      def execute(): Boolean = {
56        source.withdraw(amount)
57        target.deposite(amount)
58        executed = true
59        isExecuted
60      }
61
62      var executed: Boolean = false
63
64      def isExecuted: Boolean = executed
65
66    }
67
68    trait Bank {
69      var name: String = _
70
71      var moneyTransfers = mutable.ListBuffer.empty[Transaction]
72
73      def executeTransactions(): Unit = {
74        moneyTransfers.foreach(_.execute())
75      }
76    }
77
78    var bank: Bank = _
79
80    override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
       checkCycles: Boolean = false): BankExample = {
81      val players = (0 until numPlayer).map(i => Person("Mr.", "Stan", "Mejer" +
         i, "Fake Street 1A"))
82
83      bank = new Bank {
84        name = "Deutsche Bank"
85
86        val accounts = players.zipWithIndex.map { case (p, i) =>
87          val a = Account(Money(100.0, "USD"), i, 0.1, Money(100.0, "USD"),
             "Savings")
88          (0 until numRoles).map(ii => {
89            p addSavingsAccount a
90          })
91          a
92        }
93
94        (0 until numTransactions).foreach { _ =>
95          val source = accounts(Random.nextInt(accounts.size))
96          val target = accounts(Random.nextInt(accounts.size))
97          val transaction = Transaction(Money(10.0, "USD"), source, target, new
             DateTime)
98          moneyTransfers.append(transaction)
99        }
100     }
101     this
102   }
103
104   override def benchmark(): Unit = {
105     bank.executeTransactions()
106   }
107
108 }
```

D.3.6. SubtypeHiddenDelegation

D.3.6.1. SubtypeHiddenDelegation/BankExample.scala

Listing D.28.: Source code for SubtypeHiddenDelegation/BankExample.scala.

```
package subtypehiddendelegation

import common.{Benchmarkable, Currency => Money}
import java.util.{Date => DateTime}

import scala.collection.mutable
import scala.util.Random

class BankExample extends Benchmarkable {

  class Person(val title: String, val firstName: String, val lastName: String, val address: String)

  class Account(var balance: Money, val id: Integer) {
    def increase(amount: Money): Unit = {
      balance = balance + amount
    }

    def decrease(amount: Money): Unit = {
      balance = balance - amount
    }
  }

  class Source(id: Integer, at: Account) {
    def withdraw(amount: Money): Unit = at match {
      case sa: SavingsAccount => sa.decrease(amount)
      case ca: CheckingsAccount => ca.decrease(amount)
    }
  }

  class Target(id: Integer, at: Account) {
    def deposite(amount: Money): Unit = {
      val _ = at.increase(amount)
    }
  }

  class Customer(id: Integer, title: String, firstName: String, lastName: String, address: String) extends Person(title, firstName, lastName, address) {
    var accounts = mutable.ArrayBuffer.empty[Account]

    def addSavingsAccount(a: Account): Boolean = {
      accounts.append(a)
      true
    }

    def addCheckingsAccount(a: Account): Boolean = {
      accounts.append(a)
      true
    }
  }

  class Transaction(amount: Money, creationTime: DateTime, from: Source, to: Target) {
    def execute(): Boolean = {
      from.withdraw(amount)
      to.deposite(amount)
```

```
54        true
55      }
56    }
57
58    class MoneyTransfer(amount: Money, creationTime: DateTime, from: Source,
       to: Target) extends Transaction(amount, creationTime, from, to) {
59      var executed: Boolean = false
60
61      override def execute(): Boolean = {
62        super.execute()
63        executed = true
64        isExecuted
65      }
66
67      def isExecuted: Boolean = executed
68    }
69
70    class CheckingsAccount(balance: Money, id: Integer) extends
       Account(balance, id) {
71      var limit: Money = Money(10.0, "USD")
72
73      override def decrease(amount: Money): Unit = amount match {
74        case a if a <= limit =>
75          val _ = super.decrease(amount)
76        case _ => throw new IllegalArgumentException("Amount > limit!")
77      }
78    }
79
80    class SavingsAccount(balance: Money, id: Integer) extends Account(balance,
       id) {
81      var transactionFee: Double = 0.1
82
83      override def decrease(amount: Money): Unit = {
84        val _ = super.decrease(amount + amount * transactionFee)
85      }
86    }
87
88    trait Bank {
89      var name: String = _
90
91      var moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]
92
93      def executeTransactions(): Unit = {
94        moneyTransfers.foreach(_.execute())
95      }
96    }
97
98    var bank: Bank = _
99
100   override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
       checkCycles: Boolean = false): BankExample = {
101     val players = (0 until numPlayer).map(i => new Person("Mr.", "Stan",
         "Mejer" + i, "Fake Street 1A"))
102
103     bank = new Bank {
104       name = "Deutsche Bank"
105
106       val accounts = players.zipWithIndex.map { case (p, i) =>
107         val a = new SavingsAccount(Money(100.0, "USD"), i)
108         (0 until numRoles).map(ii => {
109           val c = new Customer(ii, p.title, p.firstName, p.lastName, p.address)
110           c.addSavingsAccount(a)
```

```
111          })
112          a
113        }
114
115        (0 until numTransactions).foreach { _ =>
116          val sA = accounts(Random.nextInt(accounts.size))
117          val tA = accounts(Random.nextInt(accounts.size))
118          val source = new Source(sA.id, sA)
119          val target = new Target(tA.id, tA)
120          val mt = new MoneyTransfer(Money(10.0, "USD"), new DateTime, source,
               target)
121          moneyTransfers.append(mt)
122        }
123      }
124      this
125    }
126
127    override def benchmark(): Unit = {
128      bank.executeTransactions()
129    }
130
131  }
```

D.3.7. SUBTYPEINTERNALFLAG

D.3.7.1. SUBTYPEINTERNALFLAG/BANKEXAMPLE.SCALA

Listing D.29.: Source code for SubtypeInternalFlag/BankExample.scala.

```
1  package subtypeinternalflag
2  
3  import common.{Benchmarkable, Currency => Money}
4  import java.util.{Date => DateTime}
5  
6  import scala.collection.mutable
7  import scala.util.Random
8  
9  class BankExample extends Benchmarkable {
10 
11   class Person(val title: String, val firstName: String, val lastName:
       String, val address: String)
12 
13   trait Account {
14     def increase(amount: Money): Unit
15 
16     def decrease(amount: Money): Unit
17   }
18 
19   class AccountImpl(var balance: Money, val id: Integer) extends
       SavingsAccount with CheckingsAccount {
20     var isCheckingsAccount = false
21     var isSavingsAccount = false
22 
23     override def increase(amount: Money): Unit = {
24       balance = balance + amount
25     }
26 
27     override def decrease(amount: Money): Unit = {
28       balance = balance - amount
29     }
30 
31     override def decreaseWithFee(amount: Money): Unit = {
32       val _ = decrease(amount + amount * transactionFee)
33     }
34 
35     override def decreaseWithLimit(amount: Money): Unit = amount match {
36       case a if a <= limit =>
37         val _ = decrease(amount)
38       case _ => throw new IllegalArgumentException("Amount > limit!")
39     }
40   }
41 
42   class Source(id: Integer, at: AccountImpl) {
43     def withdraw(amount: Money): Unit = {
44       if (at.isCheckingsAccount) {
45         at.decreaseWithLimit(amount)
46         return
47       }
48       if (at.isSavingsAccount) {
49         at.decreaseWithFee(amount)
50         return
51       }
52     }
53   }
54 
55   class Target(id: Integer, at: AccountImpl) {
```

```
56      def deposite(amount: Money): Unit = {
57        val _ = at.increase(amount)
58      }
59    }
60
61    class Customer(id: Integer, title: String, firstName: String, lastName:
        String, address: String) extends Person(title, firstName, lastName,
        address) {
62      var accounts = mutable.ArrayBuffer.empty[Account]
63
64      def addSavingsAccount(a: SavingsAccount): Boolean = {
65        accounts.append(a)
66        true
67      }
68
69      def addCheckingsAccount(a: CheckingsAccount): Boolean = {
70        accounts.append(a)
71        true
72      }
73    }
74
75    class Transaction(amount: Money, creationTime: DateTime, from: Source, to:
        Target) {
76      def execute(): Boolean = {
77        from.withdraw(amount)
78        to.deposite(amount)
79        true
80      }
81    }
82
83    class MoneyTransfer(amount: Money, creationTime: DateTime, from: Source,
        to: Target) extends Transaction(amount, creationTime, from, to) {
84      var executed: Boolean = false
85
86      override def execute(): Boolean = {
87        super.execute()
88        executed = true
89        isExecuted
90      }
91
92      def isExecuted: Boolean = executed
93    }
94
95    trait CheckingsAccount extends Account {
96      val limit: Money = Money(100.0, "USD")
97
98      def decreaseWithLimit(amount: Money): Unit
99    }
100
101   trait SavingsAccount extends Account {
102     val transactionFee: Double = 0.1
103
104     def decreaseWithFee(amount: Money): Unit
105   }
106
107   trait Bank {
108     var name: String = _
109
110     var moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]
111
112     def executeTransactions(): Unit = {
113       moneyTransfers.foreach(_.execute())
```

```
114        }
115      }
116
117      var bank: Bank = _
118
119      override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
           checkCycles: Boolean = false): BankExample = {
120        val players = (0 until numPlayer).map(i => new Person("Mr.", "Stan",
             "Mejer" + i, "Fake Street 1A"))
121
122        bank = new Bank {
123          name = "Deutsche Bank"
124
125          val accounts = players.zipWithIndex.map { case (p, i) =>
126            val a = new AccountImpl(Money(100.0, "USD"), i)
127            a.isSavingsAccount = true
128            (0 until numRoles).map(ii => {
129              val c = new Customer(ii, p.title, p.firstName, p.lastName, p.address)
130              c.addSavingsAccount(a)
131            })
132            a
133          }
134
135          (0 until numTransactions).foreach { _ =>
136            val sA = accounts(Random.nextInt(accounts.size))
137            val tA = accounts(Random.nextInt(accounts.size))
138            val source = new Source(sA.id, sA)
139            val target = new Target(tA.id, tA)
140            val mt = new MoneyTransfer(Money(10.0, "USD"), new DateTime, source,
                 target)
141            moneyTransfers.append(mt)
142          }
143        }
144        this
145      }
146
147      override def benchmark(): Unit = {
148        bank.executeTransactions()
149      }
150
151    }
```

D.3.8. SUBTYPESTATEOBJECT

D.3.8.1. SUBTYPESTATEOBJECT/BANKEXAMPLE.SCALA

Listing D.30.: Source code for SubtypeStateObject/BankExample.scala.

```
1  package subtypestateobject
2
3  import common.{Benchmarkable, Currency => Money}
4  import java.util.{Date => DateTime}
5
6  import scala.collection.mutable
7  import scala.util.Random
8
9  class BankExample extends Benchmarkable {
10
11   class Person(val title: String, val firstName: String, val lastName:
       String, val address: String)
12
13   trait Account {
14     def increase(amount: Money): Unit
15
16     def decrease(amount: Money): Unit
17   }
18
19   sealed trait AccountType extends SavingsAccount with CheckingsAccount {
20     def isSavingsAccount: Boolean = false
21
22     def isCheckingsAccount: Boolean = false
23   }
24
25   class SavingsAccountState extends AccountType {
26     override def isSavingsAccount: Boolean = true
27
28     override def decreaseWithLimit(amount: Money, account: Account): Unit = ???
29
30     override def decreaseWithFee(amount: Money, account: Account): Unit = {
31       val _ = account.decrease(amount + amount * transactionFee)
32     }
33   }
34
35   class CheckingsAccountState extends AccountType {
36     override def isCheckingsAccount: Boolean = true
37
38     override def decreaseWithFee(amount: Money, account: Account): Unit = ???
39
40     override def decreaseWithLimit(amount: Money, account: Account): Unit =
        amount match {
41       case a if a <= limit =>
42         val _ = account.decrease(amount)
43       case _ => throw new IllegalArgumentException("Amount > limit!")
44     }
45   }
46
47   class AccountImpl(var balance: Money, val id: Integer) extends Account {
48     var state: AccountType = _
49
50     override def increase(amount: Money): Unit = {
51       balance = balance + amount
52     }
53
54     override def decrease(amount: Money): Unit = {
55       balance = balance - amount
```

```
56      }
57    }
58
59    class Source(id: Integer, at: AccountImpl) {
60      def withdraw(amount: Money): Unit = {
61        if (at.state.isCheckingsAccount) {
62          at.state.decreaseWithLimit(amount, at)
63          return
64        }
65        if (at.state.isSavingsAccount) {
66          at.state.decreaseWithFee(amount, at)
67          return
68        }
69      }
70    }
71
72    class Target(id: Integer, at: AccountImpl) {
73      def deposite(amount: Money): Unit = {
74        val _ = at.increase(amount)
75      }
76    }
77
78    class Customer(id: Integer, title: String, firstName: String, lastName: String, address: String) extends Person(title, firstName, lastName, address) {
79      var accounts = mutable.ArrayBuffer.empty[Account]
80
81      def addSavingsAccount(a: Account): Boolean = {
82        accounts.append(a)
83        true
84      }
85
86      def addCheckingsAccount(a: Account): Boolean = {
87        accounts.append(a)
88        true
89      }
90    }
91
92    class Transaction(amount: Money, creationTime: DateTime, from: Source, to: Target) {
93      def execute(): Boolean = {
94        from.withdraw(amount)
95        to.deposite(amount)
96        true
97      }
98    }
99
100   class MoneyTransfer(amount: Money, creationTime: DateTime, from: Source, to: Target) extends Transaction(amount, creationTime, from, to) {
101     var executed: Boolean = false
102
103     override def execute(): Boolean = {
104       super.execute()
105       executed = true
106       isExecuted
107     }
108
109     def isExecuted: Boolean = executed
110   }
111
112   trait CheckingsAccount {
113     val limit: Money = Money(100.0, "USD")
```

```
114
115     def decreaseWithLimit(amount: Money, account: Account): Unit
116   }
117
118   trait SavingsAccount {
119     val transactionFee: Double = 0.1
120
121     def decreaseWithFee(amount: Money, account: Account): Unit
122   }
123
124   trait Bank {
125     var name: String = _
126
127     var moneyTransfers = mutable.ListBuffer.empty[MoneyTransfer]
128
129     def executeTransactions(): Unit = {
130       moneyTransfers.foreach(_.execute())
131     }
132   }
133
134   var bank: Bank = _
135
136   override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
        checkCycles: Boolean = false): BankExample = {
137     val players = (0 until numPlayer).map(i => new Person("Mr.", "Stan",
          "Mejer" + i, "Fake Street 1A"))
138
139     bank = new Bank {
140       name = "Deutsche Bank"
141
142       val accounts = players.zipWithIndex.map { case (p, i) =>
143         val a = new AccountImpl(Money(100.0, "USD"), i)
144         a.state = new SavingsAccountState()
145         (0 until numRoles).map(ii => {
146           val c = new Customer(ii, p.title, p.firstName, p.lastName, p.address)
147           c.addSavingsAccount(a)
148         })
149         a
150       }
151
152       (0 until numTransactions).foreach { _ =>
153         val sA = accounts(Random.nextInt(accounts.size))
154         val tA = accounts(Random.nextInt(accounts.size))
155         val source = new Source(sA.id, sA)
156         val target = new Target(tA.id, tA)
157         val mt = new MoneyTransfer(Money(10.0, "USD"), new DateTime, source,
              target)
158         moneyTransfers.append(mt)
159       }
160     }
161     this
162   }
163
164   override def benchmark(): Unit = {
165     bank.executeTransactions()
166   }
167
168 }
```

D.3.9. SCALADCI

D.3.9.1. SCALADCI/BANKEXAMPLE.SCALA

Listing D.31.: Source code for ScalaDCI/BankExample.scala.

```
package scaladci

import common.Benchmarkable.Backend
import common.{Benchmarkable, Currency => Money}
import java.util.{Date => DateTime}

import scala.collection.mutable

class BankExample extends Benchmarkable {

  case class Person(title: String, firstName: String, lastName: String,
    address: String)

  case class Company(POBox: String, addresses: String, legalForm: String,
    name: String)

  case class Account(var balance: Money, id: Integer) {

    def increase(amount: Money): Unit = {
      balance = balance + amount
    }

    def decrease(amount: Money): Unit = {
      balance = balance - amount
    }
  }

  @context
  case class Transaction(source: Account, target: Account, amount: Money,
    creationTime: DateTime) {

    def execute(): Boolean = {
      source.withdraw(amount)
      target.deposite(amount)
      true
    }

    role source {

      def withdraw(amount: Money): Unit = {
        source decrease amount
      }
    }

    role target {

      def deposite(amount: Money): Unit = {
        target increase amount
      }
    }

  }

  @context
  case class Bank(name: String) {
    var moneyTransfers = mutable.ListBuffer.empty[moneyTransfer]

```

```
55      def executeTransactions(): Unit = {
56        moneyTransfers.foreach(_.execute())
57      }
58
59      role customer {
60        var name: String = _
61        var id: Integer = _
62
63        var accounts = List.empty[Account]
64
65        def addSavingsAccount(a: Account): Boolean = {
66          val sa = savingsAccount
67          accounts = accounts :+ a
68          //a play sa
69          true
70        }
71
72        def addCheckingsAccount(a: Account): Boolean = {
73          val ca = checkingsAccount
74          accounts = accounts :+ a
75          //a play ca
76          true
77        }
78      }
79
80      role moneyTransfer {
81        def execute(): Boolean = {
82          moneyTransfer execute()
83          true
84        }
85      }
86
87      role checkingsAccount {
88        val limit: Money = Money(100, "USD")
89
90        def decrease(amount: Money): Unit = amount match {
91          case a if a <= limit =>
92            checkingsAccount decrease amount
93          case _ => throw new IllegalArgumentException("Amount > limit!")
94        }
95      }
96
97      role savingsAccount {
98        val transactionFee: Double = 0.1
99
100       def decrease(amount: Money): Unit = {
101         savingsAccount decrease (amount + amount * transactionFee)
102       }
103     }
104
105   }
106
107   var bank: Bank = _
108
109   override def build(numPlayer: Int, numRoles: Int, numTransactions: Int,
        backend: Backend, checkCycles: Boolean): Benchmarkable = ???
110
111   override def benchmark(): Unit = ???
112 }
```

D.3.10. OBJECTTEAMS/JAVA

D.3.10.1. OTJ/BANKEXAMPLE.SCALA

Listing D.32.: Source code for OTJ/BankExample.java.

```
import java.util.LinkedList;
import java.util.List;

public class BankExample {

  static class Person {
    private String title;
    private String firstName;
    private String lastName;
    private String address;

    public Person(String title, String firstName, String lastName, String
        address) {
      this.title = title;
      this.firstName = firstName;
      this.lastName = lastName;
      this.address = address;
    }
  }

  static class Company {
    private String POBox;
    private String addresses;
    private String legalForm;
    private String name;

    public Company(String POBox, String addresses, String legalForm, String
        name) {
      this.name = name;
      this.legalForm = legalForm;
      this.addresses = addresses;
      this.POBox = POBox;
    }
  }

  static class Account {
    private double balance;
    private int id;

    public Account(double balance, int id) {
      this.id = id;
      this.balance = balance;
    }

    public void increase(double amount) {
      this.balance += amount;
    }

    public void decrease(double amount) {
      this.balance -= amount;
    }
  }

  static team class Transaction {

    public class Source playedBy Account {
      callin void withdraw(double amount) {
```

```
56          System.out.println("Withdraw from Source");
57          base.withdraw(amount);
58        }
59
60        withdraw <- replace decrease;
61      }
62
63      public class Target playedBy Account {
64        callin void deposit(double amount) {
65          System.out.println("Deposit from Target");
66          base.deposit(amount);
67        }
68
69        deposit <- replace increase;
70      }
71
72      boolean execute(Account f, Account t, double a) {
73        f.decrease(a);
74        t.increase(a);
75        return true;
76      }
77    }
78
79    static team class Bank {
80
81      private List<Customer> customer = new LinkedList<>();
82
83      public void addCustomer(Person as Customer c) {
84        customer.add(c);
85      }
86
87      precedence SavingsAccount,CheckingsAccount;
88
89      private double limit = 100;
90      private double fee = 0.1;
91
92      public void addCheckingsAccount(Person as Customer c, Account as
          CheckingsAccount a) {
93        c.accounts.add(a);
94      }
95
96      public void addSavingsAccount(Person as Customer c, Account as
          SavingsAccount a) {
97        c.accounts.add(a);
98      }
99
100     public class CheckingsAccount playedBy Account {
101       callin void limited(double a) {
102         System.out.println("checking limit ...");
103         if (a <= limit)
104           base.limited(a);
105         else
106           throw new RuntimeException("'" + a + "' is over the limit of '" +
                limit + "'!");
107       }
108
109       void limited(double a) <- replace decrease(double a);
110     }
111
112     public class SavingsAccount playedBy Account {
113       callin void withfee(double a) {
114         System.out.println("calculating fee ...");
```

```
115          base.withfee(a + a * fee);
116        }
117
118        void withfee(double a) <- replace decrease(double a);
119      }
120
121      public class Customer playedBy Person {
122        public List<Account> accounts = new LinkedList<>();
123        private String name;
124        private int id;
125
126        public Customer(String name, int id) {
127          base("","",name,"");
128          this.name = name;
129          this.id = id;
130        }
131      }
132    }
133
134    public static void main(String[] args) {
135      // Instance level
136      Person stan = new Person("Mr.", "Stan", "Mejer", "Fake Street 1A");
137      Person brian = new Person("Mr.", "Brian", "Stephenson", "Bull Rd. 2");
138
139      Account accForStan = new Account(100.0, 1);
140      Account accForBrian = new Account(10.0, 2);
141
142      Bank bank = new Bank();
143      bank.activate();
144
145      bank.addCustomer(stan);
146      bank.addCustomer(brian);
147
148      bank.addSavingsAccount(stan, accForStan);
149      bank.addCheckingsAccount(brian, accForBrian);
150
151      System.out.println("### Before transaction ###");
152      System.out.println("Balance for Stan: " + accForStan.balance);
153      System.out.println("Balance for Brian: " + accForBrian.balance);
154
155      Transaction transaction = new Transaction();
156      transaction.activate();
157      transaction.execute(accForStan, accForBrian, 10);
158
159      System.out.println("### After transaction ###");
160      System.out.println("Balance for Stan: " + accForStan.balance);
161      System.out.println("Balance for Brian: " + accForBrian.balance);
162
163    }
164
165  }
```

LIST OF FIGURES

1.1. Ladder of technologies . . . 18

2.1. Thesis contributions and outline . . . 20

3.1. Classifying features for roles . . . 28
3.2. Role-playing automaton example . . . 34

4.1. Four-dimensional Dispatch . . . 41
4.2. Multi-dimensional Dispatch with OT/J and *SCROLL* . . . 42

6.1. The adapted Haddadin automaton . . . 47
6.2. The mapping from the Haddadin world space to the system space . . . 48

8.1. Evolving objects with roles; problem and solution . . . 56
8.2. The *SCROLL* metamodel and MOP layers . . . 57
8.3. Required basics for the implementation of a DSL for roles in structured contexts at runtime . . . 57
8.4. Example of a simple role-play graph . . . 60
8.5. Model of the robot construction as evolving object from roles . . . 61
8.6. Code for the robot construction as evolving object from roles . . . 64

9.1. The general API design of the *SCROLL* library . . . 66
9.2. Example 1 for the need of customizable role dispatch . . . 87
9.3. Example 2 for the need of customizable role dispatch . . . 88
9.4. Class diagram for the robotic co-worker example . . . 90

10.1. CompilerPlugin toolchain . . . 98

11.1. Bank example (model) . . . 112
11.2. Overall execution and build times of the bank example . . . 118
11.3. Heatmap for build times of the bank example . . . 119
11.4. Heatmap for execution times of the bank example . . . 119

13.1. Chameleon role manager . . . 128
13.2. ScalaRoles compound object approach . . . 132

14.1. The LIAM metamodel from ALIA4J . . . 144
14.2. Components and artifacts in ALIA4J . . . 144

15.1. Advance map of this thesis . . . 147

A.1. Feature model for roles (Class) . . . 155
A.2. Feature model for roles (Constraints) . . . 157
A.3. Feature model for roles (Relationships) . . . 160
A.4. Feature model for roles (Properties) . . . 161
A.5. Feature model for roles (Behavior) . . . 164
A.6. Feature model for roles (Identity) . . . 167
A.7. Feature model for roles (Lifecycle) . . . 168
A.8. Feature model for roles (Type) . . . 171

LIST OF TABLES

3.1. Ontological foundation of meta types within CROM . . . 30
3.2. Notations overview . . . 31
3.3. Role features solely focusing on runtime aspects . . . 36

5.1. Notation overview for graph traversals and filters . . . 44

7.1. Functional and non-functional requirements for *SCROLL* . . . 51

9.1. Overview for `Compartment` . . . 67
9.2. Overview for `Player` . . . 70
9.3. Overview for `DispatchQuery` . . . 72
9.4. Overview for `RoleGraph` . . . 75
9.5. Overview for `QueryStrategies` . . . 77
9.6. Overview for `RoleConstraints` . . . 78
9.7. Overview for `RoleRestrictions` . . . 79
9.8. Overview for `Relationships` . . . 80
9.9. Overview for `RoleGroups` . . . 81
9.10. Overview for `RolePlayingAutomaton` . . . 83
9.11. Overview for `SCROLLDispatch` . . . 85
9.12. Overview for `SCROLLDynamic` . . . 86
9.13. Notation overview for dispatch queries . . . 88
9.14. Mapping from dispatch functions to the Scala implementation . . . 89

10.1. The Scala compiler phases . . . 96

11.1. Comparison with contemporary approaches with regard to the requirements . 103
11.2. Qualitative evaluation of runtime role features supported by *SCROLL* . . . 106
11.3. Evaluation implementations and their full code listings . . . 113
11.4. Variation of parameters for the bank example benchmark . . . 114
11.5. The benchmark environment . . . 116

13.1. Comparison of approaches for establishing dynamic objects at runtime . . . 126
13.2. Reference role concepts and their OT/J counterparts . . . 134
13.3. Comparison of coeval approaches for etablishing roles at runtime . . . 139

14.1. Overview of the number of papers surveyed with regard to dynamic dispatch . 143

LIST OF LISTINGS

6.1 Excerpt for a plain implementation of robotic co-working example 49

8.1 Compiler rewrite rules from the `Dynamic` trait 58
8.2 Rewriting for dynamically rewritten access to the `Robot` attribute `name` 58
8.3 The generic `implicit` class `Player` 59
8.4 A naive solution for the robot example 62
8.5 A new solution for the robot example using the basic *SCROLL* API 62
8.6 Third solution for the robot example using the more advanced *SCROLL* API . 63
8.7 The `RobotExample` model source code 64
8.8 The `RobotExample` instance source code 64
8.9 The `RobotExample` console output 64

9.1 `Compartment` usage example 67
9.2 Example for using the `RolePlayingAutomaton` 84
9.3 Example code for an explicit dispatch description 88
9.4 Source code excerpt for `Machine` 91
9.5 Source code excerpt for `HaddadinAutomaton` 92
9.6 Source code excerpt for `HaddadinCompartment` 93
9.7 Source code excerpt for `SmartSensors` 94

10.1 Example for the application of the SCROLLCompilerPlugin 98
10.2 Console output of the SCROLLCompilerPlugin for the robot example 98

13.1 Code example of Chameleon 127
13.2 Code example of Rava 129
13.3 Code example of JavaStage 130
13.4 Code example of Rumer 131
13.5 Code example of ScalaRoles 132
13.6 Code example of ObjectTeams/Java 134
13.7 Code example of powerJava (role definitions) 135
13.8 Code example of powerJava (institution definition) 136
13.9 Code example of NextEJ 137

14.1 The ocean ecosystem example in Prototypes with Multiple Dispatch 145
14.2 Example for handling cartesian points with Korz 146

D.1 Source code for the `Machine` 183
D.2 Source code for `HaddadinAutomaton` 184
D.3 Source code for `HaddadinCompartment` 186
D.4 Source code for `SmartSensors` 187
D.5 Source code for `HaddadinDemo` 188
D.6 Source code for `Compartment.scala` 189
D.7 Source code for `DispatchQuery.scala` 198
D.8 Source code for `QueryStrategies.scala` 201
D.9 Source code for `Relationships.scala` 202
D.10 Source code for `RoleConstraints.scala` 204
D.11 Source code for `RoleGroups.scala` 207
D.12 Source code for `RoleRestrictions.scala` 212

D.13 Source code for RolePlayingAutomaton.scala 214
D.14 Source code for RoleGraph.scala 217
D.15 Source code for ScalaRoleGraph.scala 219
D.16 Source code for CachedScalaRoleGraph.scala 221
D.17 Source code for SCROLLErrors.scala 223
D.18 Source code for Memoiser.scala 224
D.19 Source code for ReflectiveHelper.scala 226
D.20 Source code for rop/BankExample.scala 231
D.21 Source code for rop/Component.scala 233
D.22 Source code for rop/ComponentCore.scala 233
D.23 Source code for rop/ComponentRole.scala 234
D.24 Source code for rop/scalaroles/BankExample.scala 235
D.25 Source code for SCROLL/BankExample.scala 238
D.26 Source code for SeparateType/BankExample.scala 241
D.27 Source code for SingleType/BankExample.scala 244
D.28 Source code for SubtypeHiddenDelegation/BankExample.scala 246
D.29 Source code for SubtypeInternalFlag/BankExample.scala 249
D.30 Source code for SubtypeStateObject/BankExample.scala 252
D.31 Source code for ScalaDCI/BankExample.scala 255
D.32 Source code for OTJ/BankExample.java 257

INDEX

A

acquisition dispatching 165
advanced-dispatching languages 39
around-method 35

C

callin 133
callout 133
class complexity 24
classification 23
combinatorial explosion of subtypes 25
compartment 29
compound object 27
compound object reference 164
compound object type 27
conjunctive attachment 35
constituent methods 127
consultation 27
context 29
context activation scopes 137

D

definition table 58
delegation 27
direct player reference 166
direct references 163
disjunctive attachment 169
dispatch query 88
dispatching 39
doppelgänger 168
dual self 35

E

end-user 55
explicit removal 170

F

fills 31
filters 44
final class 171
flat roles 157
forwarding 27
foundedness 30

G

generalization 23
graph 43

H

higher-order adjacency 44

I

identity 30
if-bloating 49
implicit conversion 58
implicit removal 170
indirect reference 164
institutions 135
invariants and constraints 25

L

library developer 55
lifting 169
lookup strategies 165

M

meta-functionality 171
metaclass 65
metaobject protocol 65
method handle 95
modeling 25
mono role 158
multi-role 158

N

natural 30
nested class 162
nested method 162
non-virtual self 35

O

object collaborations 24
object pooling 168
object schizophrenia 26
object-oriented model 25

P

partly frozen object state 170
play 31
playedBy 133
polymorphism 39
powers 135
private role 159
protected role 159
prototypes 155

R

receiver 27
receiver-type polymorphism 39
relationship types 30
reuse 25

rigidity 30
role as filter 36
role call 159
role creation 35
role for renaming 37
role group 32
role movement 35
role parameterization 37
role transfer 35
role-binding 33
role-playing 33
role-playing automaton 33

S

self 35
sender 27
sender-side specific dispatch 164
separation of concerns 25
singleton role 157
spaghetti reference problem 169
split-object problem 26
static method 162

T

team 133

ABBREVIATIONS

API Application Programming Interface

AST Abstract Syntax Tree

CLOS Common Lisp Object System

CROI Compartment Role Object Instance

CROM Compartment Role Object Model

DSL Domain-Specific Language

ER Entity-Relationship Model

FSM Finite State Machine

IR Intermediate Representation

JIT Just-In-Time

JRE Java Runtime Environment

JVM Java Virtual Machine

MOP Metaobject-Protocol

OT/J ObjectTeams/Java

PMD Prototypes with Multiple Dispatch

UML Unified Modeling Language

BIBLIOGRAPHY

Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *International Symposium on Handheld and Ubiquitous Computing*, pages 304–307. Springer, 1999.

Antonio Albano, Roberto Bergamini, Giorgio Ghelli, and Renzo Orsini. An object data model with roles. In *VLDB*, volume 93, pages 39–51, 1993.

Antognoni Albuquerque and Giancarlo Guizzardi. An ontological foundation for conceptual modeling datatypes based on semantic reference spaces. In *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, pages 1–12. IEEE, 2013.

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, pages 135–173. Springer, 2006.

Erik Arnaudo, Matteo Baldoni, Guido Boella, Valerio Genovese, and Roberto Grenna. An implementation of roles as affordances: powerJava. In *WOA*, pages 8–13. Citeseer, 2007.

Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I*, pages 293–334. Springer, 2006.

Charles W. Bachman and Manilal Daya. The Role Concept in Data Models. In *Proceedings of the Third International Conference on Very Large Data Bases,*, pages 464–476, Tokyo, Japan, 1977.

Stephanie Balzer, Thomas Gross, and Patrick Eugster. A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 323–346. Springer, 2007. ISBN 978-3-540-73588-5.

FSRBM Barbosa and Ademar Aguiar. Modeling and programming with roles: introducing JavaStage. Technical report, Instituto Politécnico de Castelo Branco, 2012.

Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. In *ACM SIGPLAN Notices*, volume 31, pages 122–137. ACM, 1996.

Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

Edwin Blake and Steve Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *European Conference on Object-Oriented Programming*, pages 41–50. Springer, 1987.

Daniel G Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *ACM Sigplan Notices*, volume 21, pages 17–29. ACM, 1986.

Daniel G Bobrow, Linda G DeMichiel, Richard P Gabriel, Sonya E Keene, Gregor Kiczales, and David A Moon. Common lisp object system specification. *ACM Sigplan Notices*, 23: 1–142, 1988.

Daniel G Bobrow, Richard P Gabriel, and Jon L White. Clos in Context - The Shape of the Design Space. *Object Oriented Programming: The CLOS Perspective*, pages 29–61, 1993.

Christoph Bockisch, Andreas Sewe, Haihan Yin, Mira Mezini, and Mehmet Aksit. An In-Depth Look at ALIA4J. *Journal of Object Technology*, pages 7:1–28, 2012. ISSN 1660-1769. doi: {10.5381/jot.2012.11.1.a7}. URL `http://www.jot.fm/contents/issue_2012_04/article7.html`.

Guido Boella, Steffen Goebel, Friedrich Steimann, Steffen Zschaler, and Michael Cebulla. 2'nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies, 2007.

Daniel Bonniot, Bryn Keller, and Francis Barber. The Nice user's manual, 2008. URL `http://nice.sourceforge.net/manual.html`. Accessed: 15th February 2017, 09.00.

Grady Booch. Object-oriented design with applications Benjamin. *Cummings, Redwood City (CA)*, 1991.

John Boyland and Giuseppe Castagna. Parasitic methods: An implementation of multi-methods for Java. *ACM SIGPLAN Notices*, 32:66–76, 1997.

Gilad Bracha and William Cook. Mixin-based inheritance. *ACM Sigplan Notices*, 25(10): 303–311, 1990.

Bäumer, Dirk Riehle, W. Siberski, and Martina Wulf. The Role Object Pattern. In *Washington University Dept. of Computer Science*, 1997.

Martin Büchi and Wolfgang Weck. Generic wrappers. In *European Conference on Object-Oriented Programming*, pages 201–225. Springer, 2000.

Craig Chambers. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming*, pages 33–56. Springer, 1992.

Craig Chambers. Predicate classes. In *European Conference on Object-Oriented Programming*, pages 268–296. Springer, 1993.

Craig Chambers. The Diesel Language, Specification and Rationale, 2006. URL `http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-diesel-lang/diesel-spec.pdf`. Accessed: 15th February 2017, 09.00.

Craig Chambers and Weimin Chen. Efficient multiple and predicated dispatching. In *ACM Sigplan Notices*, volume 34, pages 238–255. ACM, 1999.

Daniel Chernuchin and Gisbert Dittrich. Role types and their dependencies as components of natural types. In *2005 AAAI Fall Symposium: Roles, an interdisciplinary perspective*, 2005.

Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM Sigplan Notices*, volume 35, pages 130–145. ACM, 2000.

Antonio Cunei and Jan Vitek. PolyD: a flexible dispatching framework. *ACM SIGPLAN Notices*, 40:487–503, 2005.

Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. A generic role model for dynamic objects. In *Advanced Information Systems Engineering*, pages 643–658. Springer, 2002.

Edsgar W. Dijkstra. On the role of scientific thought, 1974. URL `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF`. Accessed: 1st December 2016, 09.00.

Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '92, pages 201–217, New York, NY, USA, 1992. ACM. ISBN 0-201-53372-3. doi: {10.1145/141936.141954}. URL `http://doi.acm.org/10.1145/141936.141954`.

Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In *European Conference on Object-Oriented Programming*, pages 253–282. Springer, 1995.

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *European Conference on Object-Oriented Programming*, pages 130–149. Springer, 2001.

Christopher Dutchyn, Paul Lu, Duane Szafron, Steven Bromling, and Wade Holst. Multi-Dispatch in the Java Virtual Machine: Design and Implementation. In *COOTS*, volume 1, pages 98–03, 2001.

Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

Torbjörn Ekman and Görel Hedin. The jastAdd Extensible Java Compiler. *ACM Sigplan Notices*, 42:1–18, 2007.

EPFL. Scala Dynamic Trait ScalaDoc, 2016a. URL `https://github.com/scala/scala/blob/2.12.x/src/library/scala/Dynamic.scala`. Accessed: 1st December 2016, 09.00.

EPFL. Scala Dynamic Trait SIP, 2016b. URL `http://docs.scala-lang.org/sips/completed/type-dynamic.html`. Accessed: 1st December 2016, 09.00.

EPFL. Scala Website, 2016c. URL `http://www.scala-lang.org/`. Accessed: 1st December 2016, 09.00.

Ericsson Utvecklings AB. Erlang Design Principles, 1999. URL `http://www.erlang.org/documentation/doc-4.8.2/doc/design_principles/fsm.html`. Accessed: 1st December 2016, 09.00.

Erik Ernst. Delegation by first-class methods. In *Welcome to the 11th Nordic Workshop on Programming and Software Develop-ment Tools and Techniques NWPER'2004 held in Turku, Finland, August 17-20, 2004. The objective of the NWPER workshops is to bring together researchers and Ph. D. students in the fields of programming and software development tools and*, page 109. Citeseer, 2004.

Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conference on Object-Oriented Programming*, pages 186–211. Springer, 1998.

Neal Feinberg, Sonya E Keene, Robert O Mathews, and P Tucker Withington. The Dylan Programming Book, 1997.

Brian Foote, Ralph E Johnson, and James Noble. Efficient multimethods in a single dispatch language. In *European Conference on Object-Oriented Programming*, pages 337–361. Springer, 2005.

I Foreman and S Danforth. *Putting Metaclasses to Work–A New Dimension in Object-Oriented Programming*. Addison Wesley, 1998.

Martin Fowler. Dealing with roles. In *Proceedings of PLoP*, volume 97, 1997.

J. Frank Furrer. Zukunftsfähige Softwaresysteme. *Informatik-Spektrum*, pages 1–9, 2015. ISSN 1432-122X. doi: {10.1007/s00287-015-0909-6}. URL `http://dx.doi.org/10.1007/s00287-015-0909-6`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

Google. Guava, 2016. URL `https://github.com/google/guava`. Accessed: 1st December 2016, 09.00.

Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems (TOIS)*, 14:268–296, 1996.

Kasper B Graversen and Johannes Beyer. *Chameleon*. PhD thesis, Masters thesis. IT-University of Copenhagen, 2002.

Kasper Bilsted Graversen. *The nature of roles*. PhD thesis, PhD thesis:/Kasper Bilsted Graversen.–Copenhagen, IT University of Copenhagen Copenhagen, 2006.

Christian Grothoff. Walkabout revisited: The runabout. In *European Conference on Object-Oriented Programming*, pages 103–125. Springer, 2003.

Sebastian Götz, Max Leuthäuser, Jan Reimann, Julia Schroeter, Christian Wende, Claas Wilke, and Uwe Aßmann. A Role-Based Language for Collaborative Robot Applications. In *Leveraging Applications of Formal Methods, Verification, and Validation*, ISoLA SARS, pages 1–15. Springer, 2011.

Sebastian Götz, Max Leuthäuser, Christian Piechnick, Jan Reimann, Sebastian Richly, Julia Schroeter, Claas Wilke, and Uwe Aßmann. Entwicklung Cyber-Physikalischer Systeme am Beispiel des NAO-Roboters. *Chemnitzer Linux-Tage - Tagungsband*, 2012.

Sami Haddadin, Michael Suppa, Stefan Fuchs, Tim Bodenmüller, Alin Albu-Schäffer, and Gerd Hirzinger. Towards the robotic co-worker. In *Robotics Research*, pages 261–282. Springer, 2011.

Terry Halpin. Object-role modeling (ORM/NIAM). In *Handbook on architectures of information systems*, pages 81–103. Springer, 2006.

Bill Harrison. Subject-oriented Programming vs. Design Patterns, 2016. URL `http://www.research.ibm.com/sop`. Accessed: 1st December 2016, 09.00.

William Harrison and Harold Ossher. *Subject-oriented programming: a critique of pure objects*, volume 28. ACM, 1993.

Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. Rava: Designing a Java extension with dynamic object roles. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 7–pp. IEEE, 2006.

James Hendler. Enhancement for multiple-inheritance. In *ACM SIGPLAN Notices*, volume 21, pages 98–106. ACM, 1986.

Rolf Hennicker and Annabelle Klarl. Helena Approach. In *Specification, Algebra, and Software*, pages 359–381. Springer, 2014.

Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Net.Object Days 2002*, October 2002.

Stephan Herrmann. ObjectTeams/Java. Technical report, AAAI Fall Symposium, 2005.

Stephan Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

Stephan Herrmann. Demystifying object schizophrenia. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '10, pages 2:1–2:5, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0535-8. doi: {http://doi.acm.org/10.1145/1929999.1930001}.

Stephan Herrmann. OTJLD version 1.3.1 Paragraph 4, 2016. URL `http://www.objectteams.org/def/1.3.1/s1.html#s1.4`. Accessed: 1st December 2016, 09.00.

Stephan Herrmann, Christine Hundt, and Katharina Mehner. Translation polymorphism in Object Teams. Technical report, TU Berlin, 2004.

Robert Hirschfeld. Aspects-aspect-oriented programming with squeak. In *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 216–232. Springer, 2002.

Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7, 2008.

Paul Hudak. Modular Domain Specific Languages and Tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE, 1998.

JetBrains. Intellij MPS, 2017. URL `https://www.jetbrains.com/mps/`. Accessed: 15th February 2017, 09.00.

Bo Nørregaard Jørgensen and Eddy Truyen. Evolution of collective object behavior in presence of simultaneous client-specific views. In *International Conference on Object-Oriented Information Systems*, pages 18–32. Springer, 2003.

Tobias Jäkel, Martin Weißbach, Kai Herrmann, Hannes Voigt, and Max Leuthäuser. Position Paper: Runtime Model for Role-Based Software Systems. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 380–387, July 2016. doi: 10.1109/ICAC.2016.17.

Karl Trygve Kalleberg, Eelco Visser, Adrian Johnstone, and Tony Sloane. Spoofax: An interactive development environment for program transformation with Stratego/XT. In *Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, pages 47–50, 2007.

Tetsuo Kamina and Tetsuo Tamai. Towards safe and flexible object adaptation. In *International Workshop on Context-Oriented Programming*, page 4. ACM, 2009.

Tetsuo Kamina and Tetsuo Tamai. A Smooth Combination of Role-based Language and Context Activation. *FOAL 2010 Proceedings*, page 15, 2010.

Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. A Comparison of Role Mechanisms in Object-Oriented Modeling. In *Modellierung*, volume 98, pages 105–109. Citeseer, 1998.

Sven Karol, Pietro Incardona, Yaser Afshar, Ivo F Sbalzarini, and Jeronimo Castrillon. Towards a Next-Generation Parallel Particle-Mesh. In van der Storm, Tijs and Erdweg, Sebastian., editor, *Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015)*, volume abs/1508.03536, pages 7–8. van der Storm, Tijs and Erdweg, Sebastian., 2015. URL `http://arxiv.org/abs/1508.03536`.

James Kempf, Warren Harris, Roy D'Souza, and Alan Snyder. Experience with CommonLoops. In *ACM SIGPLAN Notices*, volume 22, pages 214–226. ACM, 1987.

Elizabeth A Kendall. Role modelling for agent system analysis, design, and implementation. In *Agent Systems and Applications, 1999 and Third International Symposium on Mobile Agents. Proceedings. First International Symposium on*, pages 204–218. IEEE, 1999.

Setrag N Khoshafian and George P Copeland. *Object identity*, volume 21. ACM, 1986.

Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. mitpress, 1991.

Eric Kidd. Efficient compression of generic function dispatch tables. *Dartmouth College, Hanover, NH*, 2001.

SE Kleene. Object-Oriented Programming in Common Lisp, 1989.

Günter Kniesel. Objects don't migrate! Perspectives on Objects with Roles. Technical report, Universität Bonn, 1996.

Günter Kniesel. Dynamic object-based inheritance with subtyping. *PhD the*, 2000.

Bent Bruun Kristensen. Object-oriented modeling with roles. In *OOIS'95*, pages 57–71. Springer, 1996.

Thomas Kühn and Walter Cazzola. Apples and oranges: Comparing top-down and bottom-up language product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 50–59. ACM, 2016.

Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. FRaMED: Full-fledge Role Modeling Editor (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 132–136, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4447-0. doi: 10.1145/2997364.2997371. URL `http://doi.acm.org/10.1145/2997364.2997371`.

Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 141–160. Springer International Publishing, 2014. ISBN 978-3-319-11244-2. doi: 10.1007/978-3-319-11245-9_8.

Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. A combined formal model for relational context-dependent roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–124. ACM, 2015.

Joon-Sang Lee and Doo-Hwan Bae. An enhanced role model for alleviating the role-binding anomaly. *Software-Practice and Experience*, 32(14):1317–1344, 2002.

Max Leuthäuser. SCROLL - A Scala-based library for Roles at Runtime. In van der Storm, Tijs and Erdweg, Sebastian, editor, *Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015)*, volume abs/1508.03536, pages 7–8. van der Storm, Tijs and Erdweg, Sebastian, 2015.

Max Leuthäuser. SCROLLCompilerPlugin, 2016. URL `https://github.com/max-leuthaeuser/SCROLLCompilerPlugin`. Accessed: 08th May 2017, 09.00.

Max Leuthäuser. Pure Embedding of Evolving Objects. In *Proceedings of ADAPTIVE 2017, The Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications*, ADAPTIVE 2017. IARIA, 2017.

Max Leuthäuser and Uwe Aßmann. Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Establishing View-based Programming. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, MORSE/VAO '15, pages 25–33, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3614-7. doi: 10.1145/2802059.2802062.

Qing Li and Raymond K Wong. Multifaceted object modeling with roles: A comprehensive approach. *Information Sciences*, 117(3):243–266, 1999.

Lightbend Inc. Akka FSM, 2016a. URL `http://doc.akka.io/docs/akka/snapshot/scala/fsm.html`. Accessed: 1st December 2016, 09.00.

Lightbend Inc. Akka FSM, 2016b. URL `https://wiki.scala-lang.org/display/SIW/Overview+of+Compiler+Phases`. Accessed: 1st December 2016, 09.00.

Mengchi Liu and Jie Hu. Information networking model. In *Conceptual Modeling-ER 2009*, pages 131–144. Springer, 2009.

Ralf Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55. ACM, 2002.

Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive programming. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 168–177. ACM, 2003.

Bruce J MacLennan. Principles of Programming Languages: Design. -, 1983.

Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. OBJECT-ORIENTED PROGRAMMING IN THE BETA PROGRAMMING. -, 1993.

L Markovic and J Sochor. Objects with Changeable Roles. -, 2001.

Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *European Conference on Object-Oriented Programming*, pages 2–28. Springer, 2003.

Erik Meijer and August Peter Drayton. Static Typing Where Possible. *Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*, 2004.

Pottayil Harisanker Menon, Zachary Palmer, Alexander Rozenshteyn, and Scott Smith. Types for flexible objects. Technical report, Technical report, The Johns Hopkins University, 2013.

Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37:316–344, 2005.

Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

Microsoft. Expando Object, 2016. URL `https://msdn.microsoft.com/en-us/magazine/ff796227.aspx`. Accessed: 1st December 2016, 09.00.

Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *European Conference on Object-Oriented Programming*, pages 279–303. Springer, 1999.

Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *ACM SIGPLAN Notices*, volume 38, pages 224–240. ACM, 2003.

Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

David A Moon. Object-oriented programming with flavors. In *ACM SIGPLAN Notices*, volume 21, pages 1–8. ACM, 1986.

Warwick B Mugridge, John Hamer, and John G Hosking. Multi-methods in a statically-typed programming language. In *European Conference on Object-Oriented Programming*, pages 307–324. Springer, 1991.

Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *ACM SIGPLAN Notices*, volume 43, pages 563–582. ACM, 2008.

Mayur Naik and Rajeev Kumar. Efficient message dispatch in object-oriented systems. *ACM SIGPLAN Notices*, 35:49–58, 2000.

Erik Odberg. Category classes: Flexible classification and evolution in object-oriented databases. In *International Conference on Advanced Information Systems Engineering*, pages 406–420. Springer, 1994.

Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala: a comprehensive stepby-step guide. *Artima Inc, August*, 2008.

Oracle. Java Generic Types, 2015. URL `https://docs.oracle.com/javase/tutorial/java/generics/types.html`. Accessed: 15th May 2017, 14.00.

Doug Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 56–64. ACM, 2002.

Kasper Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

Kasper Østerbye. Implementation of a role language for object-specific dynamic separation of concerns. In *AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

Klaus Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *European Conference on Object-Oriented Programming*, pages 89–110. Springer, 2002.

Johan Ovlinger. *Combining Aspects and Modules*. PhD thesis, Northeastern University, 2004.

Jens Palsberg and C Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15. IEEE, 1998.

Mike P. Papazoglou and Bernd J. Kraemer. A database model for object dynamics. *The VLDB Journal—The International Journal on Very Large Data Bases*, 6(2):073–096, 1997.

Barbara Pernici. Objects with roles. *ACM SIGOIS Bulletin*, 11(2-3):205–215, 1990.

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*, pages 141–147. ACM, 2002.

Michael Pradel and Martin Odersky. A Lightweight Approach towards Reusable Collaborations. In *International Conference on Software and Data Technologies (ICSOFT'08)*, 2008.

Michael Pradel and Martin Odersky. Scala Roles: Reusable Object Collaborations in a Library. In *Software and Data Technologies*, pages 23–36. Springer Berlin Heidelberg, 2009.

Python Software Foundation. Python Enhancement Proposal 484: Type Hints, 2016a. URL `https://www.python.org/dev/peps/pep-0484/`. Accessed: 1st December 2016, 09.00.

Python Software Foundation. Python 3 Glossary on Duck-Typing, 2016b. URL `https://docs.python.org/3/glossary.html#term-duck-typing`. Accessed: 1st December 2016, 09.00.

Trygve Reenskaug and James O Coplien. The DCI architecture: A new vision of object-oriented programming. *An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html*, 2009.

Joel Richardson and Peter Schwarz. *Aspects: Extending objects to support multiple, independent roles*, volume 20. ACM, 1991.

Dirk Riehle. *Framework design*. PhD thesis, Diss. Technische Wissenschaften ETH Zürich, Nr. 13509, 2000, 2000.

Marko A Rodriguez and Peter Neubauer. The graph traversal pattern. *arXiv preprint arXiv:1004.1001*, 2010.

T Tetsuo S. Monpratarnchai. The design and implementation of a role model based language, EpsilonJ. In *Proceedings of the 5th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON 2008)*, 2008.

Markku Sakkinen. Disciplined Inheritance. In *ECOOP*, volume 89, pages 39–56, 1989.

Lee Salzman and Jonathan Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *European Conference on Object-Oriented Programming*, pages 312–336. Springer, 2005.

Michael Schrefl and Thomas Thalhammer. Using roles in Java. *Software: Practice and Experience*, 34(5):449–464, 2004.

Lars Schütze. *A Context-Aware Role-Playing Automaton for Self-Adaptive Systems*. PhD thesis, Masters thesis. Technische Universität Dresden, 2016.

Asim Anand Sinha. *Multiple Dispatch and Roles in OO Languages: Fickle*. PhD thesis, Masters thesis. Imperial College London, 2005.

Ioannis Smaragdakis. Implementing large-scale object-oriented components. -, 1999.

Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

Yannis Smaragdakis and Don S. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 550–570, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6. URL `http://dl.acm.org/citation.cfm?id=646155.679703`.

Randall B Smith and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2:161–178, 1996.

Pedro Sousa, António Rito Silva, and José Alves Marques. Object identifiers and identity: a naming issue. In *Object-Orientation in Operating Systems, 1995., Fourth International Workshop on*, pages 127–129. IEEE, 1995.

Friedrich Steimann. *Formale Modellierung mit Rollen*. PhD thesis, TU Hannover, 2000a. Habilitation thesis.

Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000b.

Lynn Andrea Stein and Stanley B Zdonik. *Clovers: The dynamic behavior of types and instances*. Brown University, Department of Computer Science, 1989.

Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education India, 1995.

Jianwen Su. Dynamic constraints and object migration. In *VLDB*, volume 91, pages 233–242, 1991.

Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys (CSUR)*, 28: 438–479, 1996.

Eddy Truyen. Dynamic and context-sensitive composition in distributed systems. -, 2004.

Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jorgensen. A dynamic customization model for distributed component-based systems. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 147–152. IEEE, 2001.

Aaron Mark Ucko. *Predicate dispatching in the common lisp object system*. PhD thesis, Massachusetts Institute of Technology, 2001.

David Ungar and Randall B Smith. *Self: The power of simplicity*, volume 22. ACM, 1987.

David Ungar, Harold Ossher, and Doug Kimelman. Korz: Simple, Symmetric, Subjective, Context-Oriented Programming. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 113–131, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1. doi: 10.1145/2661136.2661147. URL http://doi.acm.org/10.1145/2661136.2661147.

van der Torre. Roles as a Coordination Construct: Introducing powerJava. *Electr. Notes Theor. Comput. Sci*, 150(1):9–29, 2006.

Ellen Van Paesschen, Wolfgang De Meuter, and Theo D'Hondt. Domain modeling in self yields warped hierarchies. In *Workshop Reader ECOOP 2004, Oslo, Norway*, volume 3344, page 101, 2004.

Michael VanHilst. *Role oriented programming for software evolution*. PhD thesis, University of Washington, 1997.

Matthias Veit and Stephan Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 140–149. ACM, 2003.

William M Waite and Gerhard Goos. *Compiler construction*. Springer Science & Business Media, 2012.

Geertjan Wielenga. On PowerJava: "Roles" Instead Of "Objects", 2013. URL https://blogs.oracle.com/geertjan/entry/on_powerjava_roles_instead_of. Accessed: 1st December 2016, 09.00.

Yoav Zibin and Joseph Yossi Gil. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *ACM SIGPLAN Notices*, volume 37, pages 142–160. ACM, 2002.