Runtime Optimization of Contextual Role-oriented Programming Languages

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden,
Fakultät Informatik

eingereicht von

Lars Schütze

Gutachter:

Prof. Dr.-Ing. Jeronimó Castrillón-Mazo
Technische Universität Dresden

Prof. Dr.-Ing. Robert Hirschfeld
Hassno-Plattner Institut, Potsdam

Tag der Verteidigung:

18.12.2024

TECHNISCHE
UNIVERSITÄT
DRESDEN

Runtime Optimization of Contextual Role-oriented Programming Languages

Lars Schütze

21st February 2025

Für meine geliebten Kinder.
Damit sie eines Tages verstehen,
dass Informatik auch eine Wissenschaft ist.

Preamble

"The function of good software is to make the complex appear to be simple."

— Grady Booch

Abstract

Since the computer revolution, the invention of the microprocessor and the ubiquitously accessible internet software have transformed industries. This process was accompanied by the emergence of new methods and tools to handle the increasing complexity of software due to the more complex problems that computers could solve. Over the past 40 years, role-based modeling and programming have been proposed as solutions to many different technical spaces, such as data modeling, framework design, and adaptive software for cyber-physical systems.

This thesis deals primarily with role-oriented concepts in programming languages as a basis for adaptive code execution. The role-based programming paradigm extends the object-oriented paradigm. Objects may assume or discard roles that dynamically superimpose new behavior, for example, by changing the interface or binding functions to new definitions at run time. Contexts are entities that encapsulate roles and may be activated or deactivated. The roles an object plays modify its behavior as long as the context that provides the roles is active.

This thesis provides an extensive review of role-oriented programming languages since 1990. We summarize design decisions and compare implementations. From the review, we conclude efficient implementation and mapping strategies. Through benchmarking, we assess the performance of state-of-the-art implementations. By investigating their abstractions and implementation techniques we examine the semantic gap that incurs a high runtime overhead to role-oriented programming languages.

We focus on the implementation of contextual role-oriented programming language semantics, particularly on efficient execution. In most approaches the semantics is implemented via meta-object protocols. The execution of role-oriented programs uses the meta-object protocol to evaluate the runtime state. We explore the applicability of partial evaluation at different stages of compilation to increase efficiency.

First, we explore the application of partial evaluation to the evaluation algorithm of the runtime state. We discuss the approach in the context of dynamic code generation. Second, we postulate essential primitives required to bridge the semantic gap of role-oriented programming languages. We extend a virtual machine with these essential primitives, explore the optimization potential based on partial evaluation, and compare it with state-of-the-art implementations.

Publications

Some content presented in this thesis has been published previously. The following publications cited in this thesis have been authored or co-authored by me:

[SC17]	**Lars Schütze** and Jeronimo Castrillon. “Analyzing State-of-the-Art Role-based Programming Languages”. In: *Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming ’17*. Brussels, Belgium: ACM Press, 2017, pp. 1–6.
[SC19]	**Lars Schütze** and Jeronimo Castrillon. “Efficient Late Binding of Dynamic Function Compositions”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2019*. Athens, Greece: ACM Press, 2019, pp. 141–151.
[SC20]	**Lars Schütze** and Jeronimo Castrillon. “Efficient Dispatch of Multi-object Polymorphic Call Sites in Contextual Role-Oriented Programming Languages”. In: *17th International Conference on Managed Programming Languages and Runtimes*. Virtual UK: ACM, Nov. 2020, pp. 52–62.
[SKC22]	**Lars Schütze**, Cornelius Kummer, and Jeronimo Castrillon. “Guard the Cache: Dispatch Optimization in a Contextual Role-oriented Language”. In: *COP 2022: International Workshop on Context-Oriented Programming and Advanced Modularity (Collocated with ECOOP)*. Berlin Germany: ACM, June 2022, pp. 27–34.
[SC23]	**Lars Schütze** and Jeronimo Castrillon. “Towards Virtual Machine Support for Contextual Role-Oriented Programming Languages”. In: *Proceedings of the 15th ACM International Workshop on Context-Oriented Programming and Advanced Modularity (COP ’23)*. Seattle, WA, USA: ACM, July 2023.

The following publications have been authored or co-authored by me which are not part of this dissertation:

[SWD13]	**Lars Schütze**, Claas Wilke, and Birgit Demuth. “Tool-Supported Step-By-Step Debugging for the Object Constraint Language”. In: *Proceedings of the MODELS 2013 OCL Workshop*. Object Constraint Language 2013. Miami, USA, Sept. 30, 2013.

[Kho+21] Nesrine Khouzami, **Lars Schütze**, Pietro Incardona, Landfried Kraatz, Tina Subic, Jeronimo Castrillon, and Ivo F. Sbalzarini. “The OpenPME Problem Solving Environment for Numerical Simulations”. In: *Computational Science – ICCS 2021*. Vol. 12742. Cham: Springer International Publishing, 2021, pp. 614–627.

[SSK22] Ilja Shmelkin, **Lars Schütze**, and Tim Kluge. “Modeling Flexible Monitoring Systems with a Role-Based Control Loop”. In: *COP 2022: International Workshop on Context-Oriented Programming and Advanced Modularity (Collocated with ECOOP)*. COP ’22: International Workshop on Context-Oriented Programming and Advanced Modularity. Berlin Germany: ACM, June 7, 2022, pp. 18–26.

[Gün+23] Mirko Günther, **Lars Schütze**, Kilian Becher, Thorsten Strufe, and Jeronimo Castrillon. *HElium: A Language and Compiler for Fully Homomorphic Encryption with Support for Proxy Re-Encryption*. Dec. 21, 2023. arXiv: `2312.14250 [cs]`. URL: `http://arxiv.org/abs/2312.14250` (visited on 01/03/2024). preprint.

Acknowledgements

It was Birgit Demuth and Claas Wilke who in large part are responsible for introducing me to the joy, humility, and gratitude of research. After many interesting discussions about the contents of the lecture on Software Technology Birgit recruited me for her research project `Dresden OCL` in the 2$^{\text{nd}}$ semester of my studies of Computer Science. During that time I learned a lot about modeling, the design and implementation of modeling languages, and everything that is meta-*, e.g., meta-models, meta-meta-models. All this knowledge was very useful when taking on the project that became my PhD. At some point, it is a unique skill to be able to understand the different aspects of a programming language from various points of view — the meta-model that describes the language concepts, the relation of the model to the syntax and the language semantics, and the many aspects of the implementation itself.

I want to thank Prof. Aßmann whom I owe for developing a deep understanding of software design, roles, and technical spaces due to his series of lectures that I visited during my master's program. He introduced me to the research program RoSI (Role-based Software Infrastructures for continuous-context-sensitive Systems) which later accepted my application and financed my research. Special thanks go to Thomas Kühn and Sven Karol who were a role model of a researcher to me and later became colleagues and mentors. I want to also thank my former colleagues from the Chair of Software Technology and RoSI including Max Leuthäuser, Johannes Mey, Johannes Bamme, Christian Piechnick, and Nguonly Taing for all the interesting discussions.

I am grateful to my supervisor Jeronimo Castrillon who supported me over many years. He gave me complete freedom and trust to pursue the research topic and was always available to give constructive feedback on the many ideas I had. He simultaneously encouraged and challenged me to continuously enhance my research, writing, and presentation skills. Thanks to him, I have developed a deep understanding of compilers and have been entrusted with teaching the seminar on compiler construction to our students which I enjoyed. The environment he established at the Chair for Compiler Construction enabled me to become an independent and confident researcher who is prepared to tackle new challenges. I have always enjoyed the international atmosphere and working with people from around the world at the Chair for Compiler Construction. Thanks to my colleagues for all the interesting discussions we had about just anything. In particular, I want to highlight Nesrine Khouzami and Andrés Goens for all the interesting discussions about many aspects of life, and Christian Menard, Felix Suchert, and Karl Friebel for all the technical discussions.

I also want to thank my parents for their support throughout my whole life. They accepted all the life-changing choices I made and unconditionally supported them. I would not have achieved all of that without you.

But nothing would have gotten me to the stage of writing these lines if it were not for my beloved wife Melanie who endured the work at late nights, during the weekend, and vacations. Thanks for all the support and our two lovely kids. Even when they pretend that sitting in front of a computer does not count as research.

Contents

Part I

Introduction and Background

1 Introduction

"The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."
— Edsger W. Dijkstra (EWD340)

The human mind can think on many levels of abstraction. By creating abstractions one encapsulates the details and creates well-defined boundaries. This enables us to understand complex scenarios and interactions and to reason about them. At the same time, it is the boundaries that need to be crossed to create dynamically adaptable software. This Chapter introduces the problems that have been addressed by well-known approaches such as Object-oriented Programming. We then introduce the domain of dynamically adaptable software where Object-oriented Programming struggles to deliver satisfactory solutions and how its extension with roles—Role-oriented Programming—provides all the necessary properties. We define the problem responsible for runtime overheads in Role-oriented Programming and the dissertation's contributions. Last, the content of the dissertation is outlined.

1.1 Reusable and Interchangeable Software Components

In his well-known article "Why Software Is Eating The World" [And11] Marc Andreessen covers how many economic fields were disrupted by software. He relates how the technologies of the computer revolution, the invention of the microprocessor, and the ubiquitously accessible internet made software viable to transform industries.

All this has been possible, because, at the same time, the methods to develop software and the programming paradigms evolved, too. The reason is the increase in the complexity of the software systems based on the needs of the market or problems that they were trying to solve. For example, in the late 1960s, truly large software systems were attempted to be built commercially–the OS 360 operating system for the IBM 360 computer family was one of them. It was realized that building large software systems was significantly different from building smaller systems. The idea was to approach the problem of building software in the same way that engineers used to build large complex systems. Thus, software engineering was born [GJM03].

Dijkstra stated the *software crisis* existed because of the missing methods and tools [Dij72]. They were inadequate to take on the increasing complexity of the problems that computers could solve. In general, the complexity of a software system results from the problem domain being inherently complex and naturally requiring a complex solution [SVL08]. The task is to manage the *essential complexity* and to avoid *accidental complexity.*

Building software systems was not only impacted by the organizational aspects introduced by software engineering but also the programming languages used in software development. They have a profound influence on how well the software engineering goals can be reached. For instance, *modularity* features such as separate specification and implementation supported the development of large software systems in teams. Ada 95, for example, divides a software system into *packages.* Libraries of packages can be separately developed, deployed, and consumed as *components.*

> "What does is realizing that *encapsulated* complexity is no longer complexity at all. Its gone, buried forever in somebody else's problem. This shifts the focus to the human **systems** that manage complexity successfully versus those that don't."
>
> – Brad Cox [Cox94]

It was the object-oriented (OO) paradigm that provided technical solutions to the software crisis and allowed a paradigm shift towards *reusable* and *interchangeable* software components [Cox95]. At its core were the principles of abstraction, encapsulation, generalization, and polymorphism. As Cox states, encapsulation made it possible to hide a complex problem and make it accessible through interfaces.

Object-oriented Analysis (OOA) helped to understand and capture the problem domain and offered tools to communicate among stakeholders such as domain experts and software engineers. The Unified Modeling Language (UML) became one of the main tools providing well-defined models and meta-models as a means to communicate; and later to generate code. Object-oriented Design (OOD) provided a final model solution that required a mere implementation. This built the foundation for subsequent research in software engineering that ultimately resulted in one of the seminal books on principals of reusable object-oriented software [Gam+95].

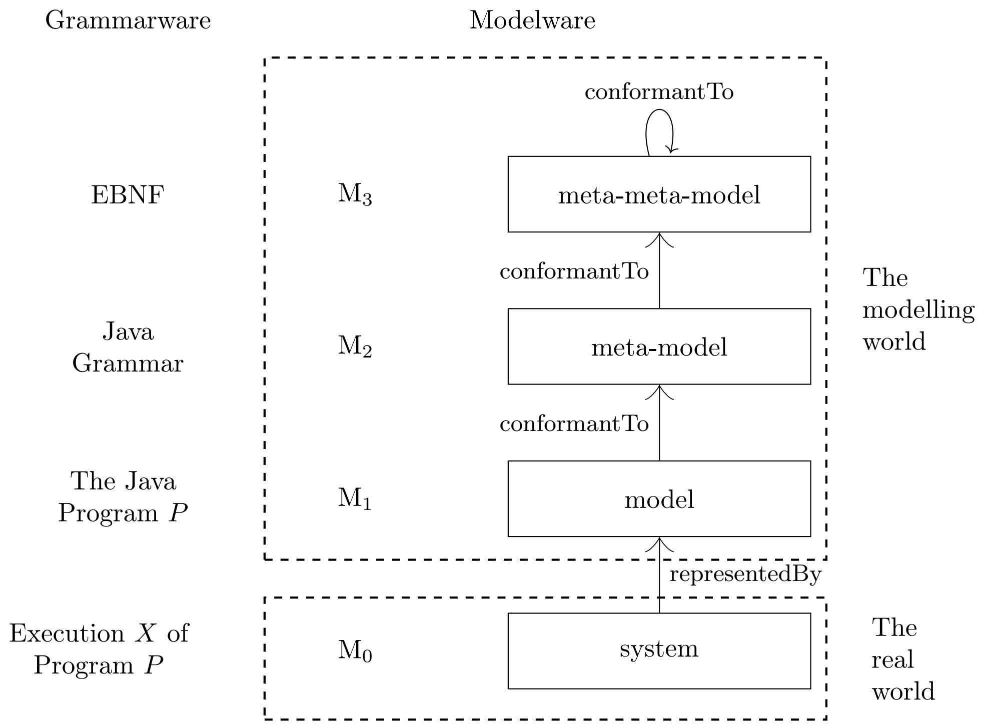


Figure 1.1: The technical spaces modelware and grammarware and the meta-level hierarchy.

Figure 1.1 highlights the importance of advances in Model-driven Software Development (MDSD) because of the relation between programming languages and meta-models. The technical spaces of grammarware and modelware can be aligned in a meta-level hierarchy [FD99] where for each level in the hierarchy the technical spaces co-relate. For example, consider a UML class diagram to be a model that is conformant to the UML meta-model. It can represent the static structure of a specific Java program P, that is at the same meta-level, which is conformant to the Java grammar.

1.2 Software Adaptability Problem

To organize object-oriented programs there are two key principles: One is to identify and categorize similar objects into classes known as *classification*. The other is how relationships of classes are organized into a class hierarchy using *generalization*, i.e., sharing common behavior and attributes in a superclass. Classes encapsulate the implementation and provide interfaces. Object-oriented Programming (OOP) excels at representing the structure of a domain but struggles to represent dynamic collaborations of objects over time. In most class-based object-oriented systems the association between an instance of a class and the class itself is permanent [GSR96]. In such systems, the requirements are set and can hardly be adapted at runtime. Thus, class hierarchies need to be carefully planned to support (dynamic) extensions [Fow97]. However, in real-world systems, it was noticed that objects tend to change their types and associations over time [AGO93]. The inflexibility of the association of instances to their classes becomes a problem in a dynamic, frequently changing environment, where objects need to change accordingly. Capturing the semantics of context-dependent adaptations is a verbose and error-prone task. In such a system, the application of object-oriented patterns [Gam+95] to reduce coupling and improve cohesion constitutes an overhead on the runtime and may introduce subtle errors when not implemented carefully.

The importance of roles during OOA has been pointed out by Pernici [Per90], Papazoglou [Pap91], and Morton and Odell [MO92]. Roles are analogous to human conceptual understanding and have been naturally found in data models [BD77] and conceptual modeling [RWL96]. Role-oriented Programming (ROP) is a programming paradigm where *roles* encapsulate behavior. The association of an instance of a class to the class itself is not as rigid as in OOP. Objects can adapt to changes in their environment, or context, by assuming roles. Role-oriented programming languages achieve this by providing classes, roles, and contexts as first-class citizens. So far, they have provided the most natural solution to the adaptivity problem.

Since their inception over four decades ago [BD77], research focused on roles in different contexts. This thesis is concerned with the research on roles in programming languages. In 2000 - 2005 and beyond, *context* (or *compartment*) was introduced as a new conceptual entity. Roles became aware of the context they are defined within and in which they must be used. This tamed the increased complexity of contemporary context-sensitive, distributed software systems.

1.3 Problem Definition

The previous section discussed that contextual roles can balance the complexity of current context-sensitive, distributed software systems. To implement the concept of contextual roles, approaches explored patterns of delegation and forwarding where multiple instances represent parts of a conceptual whole. Solutions based on patterns [Bäu+97; Fow97] had undesired properties such as *object schizophrenia* [CJ02].

It is noticed that compiler-based approaches, among other benefits, could work around object schizophrenia [Her10].

The issue is that implementation strategies produce a verbose description of contextual roles in an object-oriented execution environment which incurs an overhead. The context-dependent nature of roles impedes a direct mapping of role fulfillment into class hierarchies since the introduction of contexts as first-class citizens, increasing the need for compilers and optimizations. Since the object-oriented target machine model does not understand aspect semantics role-oriented concepts are emulated using multiple instructions. However, approaches did not take into account whether compilers and language runtimes could optimize the resulting programs. In consequence, the resulting *semantic gap* produces a runtime overhead of managing the dynamic relations between objects and the roles they play in different contexts.

Thus, this thesis makes the following contributions to contextual, role-based programming languages and implementations:

1. It surveys past and contemporary literature on role-oriented programming languages, their features, and implementation strategies (Chapter 5).
2. It quantitatively analyzes contemporary role-oriented programming languages using different benchmarks, highlighting the need for optimizations (Chapter 6).
3. A discussion about the *semantic gap* resulting from mapping role-oriented programs to object-oriented programming languages (Chapter 7).
4. Dispatch Graphs, a runtime-generated, graph-based approach is proposed to overcome the semantic gap and *enable compiler optimizations* to role dispatch (Chapter 8).
5. It proposes a Virtual Machine (VM) implementation of contextual roles which closes the semantic gap in Chapter 9.

1.4 Outline

We consider (contextual) Role-oriented Programming not only as an extension of Object-oriented Programming but as an independent, advanced programming concept that lives on its own. Thus, we divided this thesis into three parts as shown in Figure 1.2.

The **first part** introduces the many concepts the thesis is built on. Chapter 2 introduces the different approaches of method resolution and late binding referenced

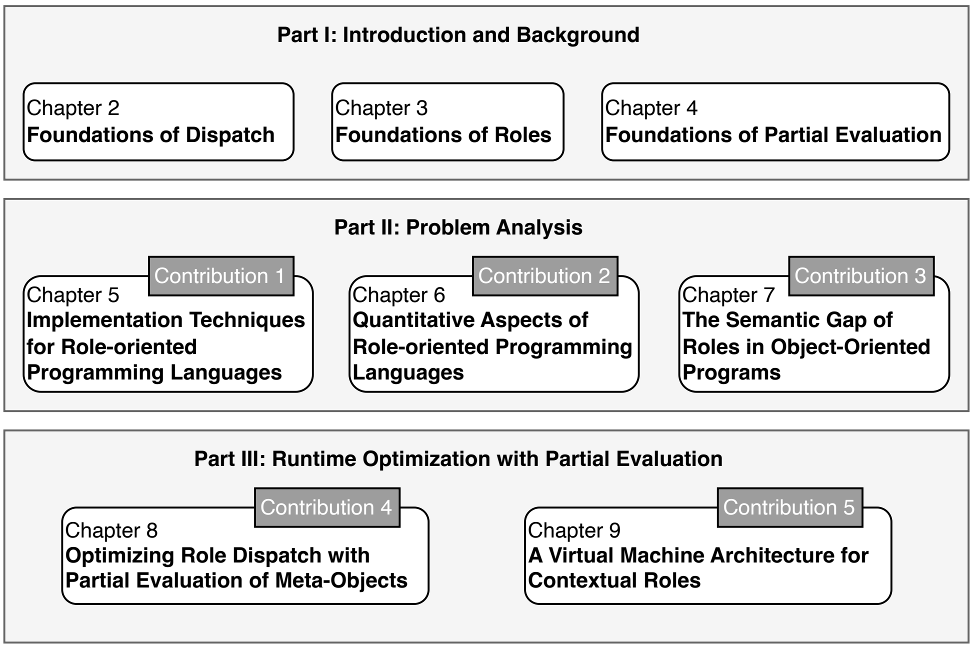

Part I: Introduction and Background
Chapter 2
Foundations of Dispatch
Chapter 3
Foundations of Roles
Chapter 4
Foundations of Partial Evaluation
Part II: Problem Analysis
Contribution 1
Chapter 5
Implementation Techniques for Role-oriented Programming Languages
Contribution 2
Chapter 6
Quantitative Aspects of Role-oriented Programming Languages
Contribution 3
Chapter 7
The Semantic Gap of Roles in Object-Oriented Programs
Part III: Runtime Optimization with Partial Evaluation
Contribution 4
Chapter 8
Optimizing Role Dispatch with Partial Evaluation of Meta-Objects
Contribution 5
Chapter 9
A Virtual Machine Architecture for Contextual Roles

Figure 1.2: Overview of this Dissertation.

throughout the thesis. In Chapter 3 we introduce the different related approaches and give a deep introduction to Role-oriented Programming. Partial evaluation, as one of the main concepts applied by this thesis, is introduced in Chapter 4.

The **second part** reviews and analyses past and contemporary Role-oriented Programming approaches. Chapter 5 surveys Role-oriented Programming languages published later than 1990. Surveyed approaches are classified according to their role features and compiled into a list of implementation strategies. To discover and explain performance characteristics of Role-oriented Programming languages a quantitative analysis is conducted in Chapter 6. The discovered semantic gap is explained in Chapter 7 based on a description of the Metaobject Protocol of a contemporary Role-oriented Programming language used to represent features ascribed to roles in Object-oriented Programming paradigms.

The **third part** proposed solutions to enable optimizations of Role-oriented Programming languages at runtime. In Chapter 8 a reference object model of Role-oriented

Programming is introduced. The model allows us to describe role dispatch in detail abstracting from a concrete implementation and to define optimizations based on partial evaluation at runtime. Chapter 9 proposed a Virtual Machine architecture and Abstract Syntax Tree interpreter to enable partial evaluation of dispatches in Role-oriented Programming.

Last, related work is discussed and the thesis is concluded.

2 Foundations of Dispatch

"Language shapes the way we think, and determines what we can think about."

— Benjamin Lee Whorf

This chapter starts with a brief introduction to the basic mechanisms of resolving function calls and their usage in Multi-Dimensional Separation of Concerns (MDSoC) approaches. The goal is to develop an understanding of the similarity of the problems the approaches were set out to solve and the introduction of a common terminology used in this thesis. The distinction is also necessary to undermine the fact that Role-oriented Programming is a paradigm in its own right that has its root in domain modeling for object-oriented databases and object-oriented analysis. The presented implementations provide unique solutions to the problems related to dealing with roles.

2.1 Models of Dispatch

A key design feature of contemporary programming languages is their support for some kind of modularization, such as classes, namespaces, and packages. Advanced modularization support beyond these basic capabilities, more often than not, are transformed into core language constructs of their host languages, i.e., the languages that the language extension or Domain-specific Language (DSL) translates or compiles to. The realization may have a severe impact on the resulting run-time performance. It is the responsibility of the compiler, language implementation, or runtime to consider performance as another quality metric for these transformations.

A mechanism to increase modularity is *polymorphism* where a concrete module must not declare (or refer to) another *concrete* module, but only abstractly specify the reference. Types of polymorphism range from receiver-type polymorphism that is commonly used in OOP languages nowadays, to multiple dispatch [KG89] and predicate dispatch [EKC98], over pointcut-advice in Aspect-oriented Programming (AOP) [MK03], and layered methods in Context-oriented Programming (COP) [HCN08].

The task of the *dispatching* mechanism is to resolve, i.e., lookup, the polymorphic code locations to concrete procedures that will be executed. The resolution can be done statically at compile-time using the static type or at run-time using the dynamic type of one or more arguments upon method invocation. Often the resulting procedure is bound to the code location and reused when the same argument types are seen again. Retaining multiple mappings from argument types to resolved procedures at code locations is an optimization called Polymorphic Inline Cache (PIC) [HCU91].

Implementations of role-oriented programming languages often build on established programming languages, using existing concepts and implementations as their intermediate representations. In the following, we will introduce the main dispatching mechanisms used in class-based object-oriented programming languages since they govern implementation techniques used in role-oriented programming languages. Depending on the desired features of the role model different dispatching semantics are favored.

2.1.1 Single Dispatch

In the Simula dispatching model [DN66], the first argument in a procedure call is implicitly the receiver—the object's dynamic type determines the concrete procedure executed. This model is inherited by Smalltalk [GR89] which is a precursor to today's many object-oriented programming languages, such as C++ and Java. In object-oriented *class-based* approaches methods are associated with classes in a class hierarchy or partially ordered set. Given that encapsulation is provided in some form, i.e., records, or classes, this yields the benefits of abstract data types and provides the receiver-type polymorphism. The model is well aligned with decompositions following the class hierarchy but unaligned with problems involving multiple associated classes (or class hierarchies), a problem well-known to several classical problems in Object-oriented Programming.

2.1.2 Multiple Dispatch

Using the dynamic type of more than one argument to decide the procedure that is executed is called multiple dispatch. There exist multiple programming languages that implement multiple dispatch such as CLOS [KG89], Cecil [Cha92], Diesel [Cha98], MultiJava [Cli+00], and more [Sha96; MHH91; PSS07; CM95; Bez+12]. A procedure that uses multiple dispatch is called multimethod. Multimethods may be defined as external to the classes they are concerned with and are not encapsulated resulting in *open classes*. One method overrides another method if its specializer classes are subclasses of the other's using either lexicographic (CLOS [KG89]) or pointwise (Cecil [Cha92]) ordering.

Multiple dispatch can solve problems occurring with object-oriented design patterns.

For instance, the visitor pattern [Gam+95] requires a dispatch over the concrete type of the visitor as well as the concrete type of the visited object. With multiple dispatch, multimethods can be defined for each case while in single dispatch this will require a generic procedure that uses `if` and `instanceof` to delegate to the concrete visited type. This phenomenon is also called double dispatch [Mus+08].

Role-oriented Programming defines two related class hierarchies where one is of the role-playing entities and the other is of the roles these entities play. Chimera [CM95] uses multi-method dispatch (*argument specificity approach*) for role dispatch. In [Kni96, p.19] it is discussed that multi-method dispatch without open classes does not allow dispatch over context-dependent behavior.

2.1.3 Predicate Classes and Predicate Dispatch

The concept of Predicate Classes [Cha93] involves the automatic classification of an object as a subclass of a predicate class A based on the object satisfying a predicate expression associated with A. The predicate expression can test the value or state of the object. By linking methods to predicate classes, method lookup can depend not only on the object's class but implicitly on its dynamic value or state. This approach enables method dispatching even when an object's effective class may change over time.

In single and multiple dispatch, the overriding relationship depends on whether a subclass defines a same-named method with the same argument types[1]. In predicate dispatching arbitrary predicates and logical implications between predicates may define the overriding relationship [EKC98]. A method m_2 overrides another method m_1 if the predicate of m_2 implies m_1. Predicate dispatch subsumes the aforementioned principles of single and multiple dispatch as well as facilities known from pattern matching as in ML [Mil97] and Haskell [Hud+92].

Since predicate expressions can test the value or state of the object, one of the principles of object-oriented design–encapsulation–is undermined. An example extracted from [EKC98] is shown in Listing 2.1 which shows the implementation of an optimizing compiler using a programming language with predicate dispatch. In that example, the default method of `ConstantFold` is overridden when the predicate holds. The predicate is that the expression `e` is of type `BinopExpr` and its properties fulfill a specific type, i.e., the operation is of type `IntPlus` and the arguments are of type `IntConst`.

[1]The Liskov substitution principle defines that a function $f_1 : A_1 \rightarrow B_1$ is replaced by a function of a sub-class $f_2 : A_2 \rightarrow B_2$ when it holds that $B_2 \leq B_1$ (covariant return type) and $A_1 \leq A_2$ (contravariant argument type). Some languages relax the model. For instance, Java does not support contravariant argument types in overridden functions

Listing 2.1 Predicate dispatch example. Adapted from [EKC98].

```
type Expr;
  signature ConstantFold(Expr):Expr;
  -- default implementation
  method ConstantFold(e) { return e; }

type Binop;

class BinopExpr subtypes Expr {
  op: Binop, arg1: AtomicExpr, arg2: AtomicExpr };
  -- override default to constant fold binops with const args
  method ConstantFold(
    e@BinopExpr { op@IntPlus, arg1@IntConst, arg2@IntConst } ) {
    return new IntConst { value := arg1 + arg2 };
  }
```

2.1.4 Multi-Dimensional Dispatch

In single dispatched object-oriented languages the lookup of a method is rather rigid. Multiple dispatch can encode the selection of different methods based on the type of receiver and the sender. Smith et al. [SU96] argue that it needs a more fine-grained way to select appropriate functions based on the sender-receiver relation for each message send[2], an approach called Subject-oriented Programming (SOP) [HO93]. Smith et al. form the idea of the dimensionality of method dispatch, contemplating how subjective programming forms a third dimension, where object-oriented languages form the second dimension, and procedural languages the first. This idea has since been extended to a fourth dimension also taking the *context* of a message into account [HCN08]. Figure 2.1 presents the idea of Hirschfeld et al. and shows the different dimensions and the parts they are concerned with. The arrows represent messages sent and their resolution schematically. In this thesis, we will investigate the semantics expressed by these arrows and their implementations in Context-oriented Programming and Role-oriented Programming languages.

[2]In the object-oriented language SELF [CUL89] dispatching is represented by objects sending messages to other objects.

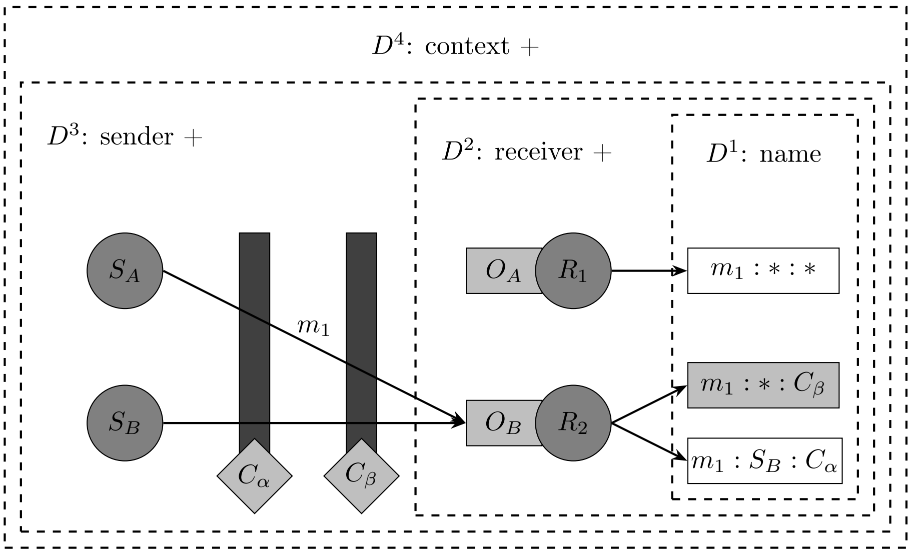


Figure 2.1: Four-dimensional dispatch, adapted from [HCN08, Fig.3].

2.2 Concepts of Object Inheritance

In Chapter 2.1 we introduced the foundations of dispatch in class-based object-oriented programming languages. Role-playing and its implementation are all about the *communication of objects*. Prototype-based object-oriented programming languages played a key role in early role-based programming language research as exchanging messages between objects is a fundamental concept of object-based OOP.

Inheritance introduces the methods from the super-object into the inheriting object. When invoking a method of the super-object the question is to what object `self` is bound.[3] We found different approaches to object inheritance in the literature, each serving a different purpose. These approaches represent the type of object-oriented programming language they are employed in. Delegation and inheritance are mechanisms for sharing in object-oriented systems [Ste87]. For example, *delegation* is a mechanism commonly used in prototype-based object-oriented programming languages as an inheritance mechanism. In contrast, *forwarding* is often used in class-based inheritance. A taxonomy of prototype-based object-oriented programming languages can be found in [DMB98]. We want to note that being classless and providing a delegation mecha-

[3] `self` describes the implicit self-reference of an object or instance. In C++ or Java it is known as `this`.

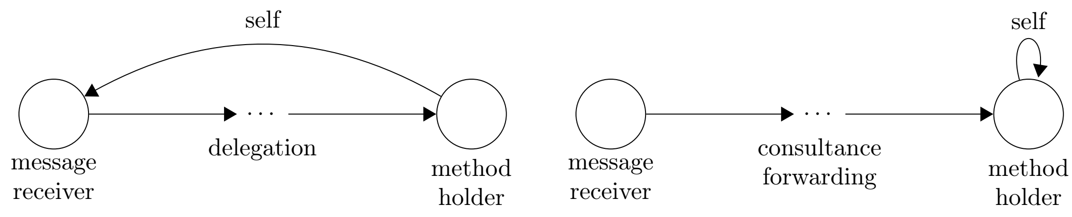


Figure 2.2: The effect of forwarding, consultation, and delegation on `self`. Figure adapted from [KRC91, Fig.1].

nism are orthogonal features of object-oriented programming languages [Mal95]. Nevertheless, delegation-based approaches exist in class-based, single-dispatched object-oriented programming languages [RO13]. In the following, we will present each mechanism so the implications of approaches using them become clear.

2.2.1 Fowarding and Consultation

Inheritance introduces the methods from the super-object into the inheriting object. Although a subclass has an explicit *super*-object, in the presence of forwarding, requesting super is not the same as making a direct request to the super-object, as the value of `self` will be bound to the inheriting object. Under forwarding (also termed consultance [KRC91]), inherited methods are redirected to the super-object. By *early-binding* the value of `self` when an object is created in both its methods and field accessors the super-method receives the same arguments and `self` binding. The result of this change is that the late-binding of `self` in requests (both normal and to super) no longer achieves anything because `self` has already been bound to the object that the method or field originally appeared in [Jon+16]. This also implies that forwarding does not permit down-calls; to achieve down-calls from the inherited object the inheriting object must explicitly be passed as a parameter. [4]

2.2.2 Delegation

In class-based object-oriented programming, classes must be created before their instances can be used and behavior is only associated with classes. This relation fixes

[4]Be reminded that we discussed object-based inheritance. In the class-based object-oriented programming paradigm, the super-calls will be bound to `self` permitting down-calls from the inherited to the inheriting object.

the communication pattern of objects at instance creation time. The delegation mechanism, as it is defined in [LSU88; Lie86] is not exactly a message redirection but a kind of inheritance mechanism that can be understood in a way as message redirection. Delegation in this sense is thus a kind of inheritance mechanism that is commonly associated with prototype-based programming [LSU88; US87; Weg87]. The delegation mechanism is based on a link (generally called *parent* or *delegation* link) between entities holding values, i.e., objects or classes. There is a subtle distinction from forwarding: a `self` request in a method called under delegation goes back to the original object, while under forwarding a `self` request to a delegatee will be handled only by that delegatee [Jon+16]. Because any object can be used as a prototype and messages can be retargeted to other objects at any time, delegation is the more flexible, dynamic, and general approach of the two techniques.

2.3 Meta-Object Protocols and Open Implementation

There exist multiple object-oriented design patterns to realize possible extensions of a program either concerned with the extension/adaptation of single entities or groups of entities [Gam+95]. However, patterns are restricted by their expressiveness. Crosscutting extensions of unrelated entities often lead to extension logic being scattered across the program. The idea of a Metaobject Protocol (MOP) is to provide an interface to the language that "*give the users the ability to incrementally modify the language's behavior (semantics) and implementation, as well as the ability to write the programs with the language*" [Pae93]. In a language based upon Metaobject Protocols, the language implementation itself is structured as an object-oriented program [KDB91]. A MOP is composed by a set of entry points, i.e., (abstract) functions, whose specialization allows the introduction of new behavior. In general, MOPs are based on meta-objects offering ways of specializing their behavior and representation specific aspects of the base level. This may be realized using reflective programming capabilities of the programming language itself (such as realized in Common Lisp Object System (CLOS) [GWB91]) or at the application level itself in object-oriented class hierarchies as realized in many modern object-oriented programming languages. MOPs may be used to realize *open implementation* principles where meta-objects of the system can be used to define the structure and static properties and their protocols are extended, i.e., the dynamic behavior of these objects are adapted.

3 Foundations of Roles

"All information that can be expressed in a class-based model can be expressed in a role model. All information that can be expressed in an object-based model can be expressed in the same role model."
— T. Reenskaug, P. Wold, and O. Lehne, 1996

This chapter starts with a brief introduction to the basic mechanisms of resolving function calls and their usage in MDSoC approaches. The goal is to develop an understanding of the similarity of the problems the approaches were set out to solve and the introduction of a common terminology used in this thesis. The distinction is also necessary to undermine the fact that Role-oriented Programming is a paradigm in its own right that has its root in domain modeling for object-oriented databases and object-oriented analysis. The presented implementations provide unique solutions to the problems related to dealing with roles.

3.1 Multi-Dimensional Separation of Concerns

Separation of Concerns (SoC) is a universal approach to managing large and complex software systems. Reduced complexity and increased comprehensibility require some *decomposition* mechanism to divide software into meaningful and manageable artifacts. It also requires *composition* mechanisms to assemble the software artifacts. Most contemporary programming languages provide these mechanisms. However, in general, decomposition and composition are only supported along a single domain, e.g., by function, object, or class. This "Tyranny of the dominant decomposition" [Tar+99] is a constraining factor in developing context-dependent software that must decompose functionality among multiple dimensions.

Design patterns improve the extensibility and adaptability of software architectures and often they directly impact the overall quality. However, they do not provide a sufficient solution to implement cross-cutting concerns resulting in tangled and scattered context-dependent code fragments. Moreover, object-oriented programming languages do often fail to express solutions to the expression problem [ZO04]. The expression problem arises when recursively defined data types and operations defined on them have

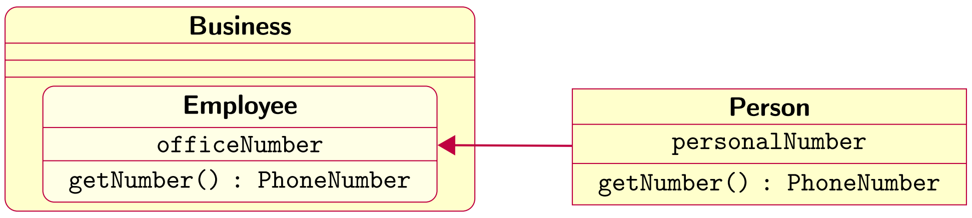


Figure 3.1: A role model of the office number example using Compartment Role Object Model (CROM) [Küh+14]. In the context of a `Business`, a `Person` behaves as `Employee`.

to be extended simultaneously. To solve these problems, different Separation of Concerns approaches have been proposed with varying degrees of granularity ranging from adapting single objects or functions to classes and components [OT99; Aßm03]. To enable *dynamic* SoC approaches focus on the *interaction* of classes, such as in Subject-oriented Programming (SOP) [HO93], Aspect-oriented Programming (AOP) [Kic+97], and Context-oriented Programming (COP) [HCN08]. This enables us to decompose the application into different views and concerns during development and to compose them at runtime adaptively. The result is a flexible software development process and an adaptive execution model.

In the following, we will use the case of a business number as a running example to introduce the different concepts. The role model representing the use case using Compartment Role Object Model [Küh+14] is shown in Figure 3.1. The system consists of an entity `Person` with the defined behavior to hand out his private phone number. Given a change in the context, for instance, at work, the person's behavior will adapt and return an appropriate phone number from his office. The base application is depicted as a snipped of Java code in Listing 3.2. Concepts introduced in the following sections will present implementations based on this scenario.

3.1.1 Crosscutting Concerns and the Aspect-oriented Paradigm

In Aspect-oriented Programming (AOP) [Kic+97] *cross-cutting concerns* are decomposed into aspects that would be otherwise scattered over the code and tangled with the application logic. Masuhara et al. devised a framework to classify different models of Aspect-oriented Programming [MK03]. They identified four different flavors that each apply a different type of crosscutting: pointcuts and advice, traversal specifications, class compositions, and open classes. We focus on the pointcuts and advice model. *Aspects* encapsulate such cross-cutting concerns and provide expressions to define in-

Listing 3.2 Base application for the running example.

```
class Person {
  private PhoneNumber number;
  public Person(PhoneNumber number) {
    this.number = number;
  }

  PhoneNumber getNumber() {
    return number;
  }
}
```

terceptors, class extensions (inter-type declarations), and their properties [Bri+05]. An aspect can alter the behavior of non-aspect parts of the program, called *base class*, by applying the advice that defines the additional behavior. Alternative behavior can be applied at *join points* in the base program including function calls and property access. The conditions when behavioral adaptations are applied are expressed in *pointcuts* providing predicates that quantify the set of existing join points and choose the set of join points where the execution of the advice is desired. To let applications use the specified aspects they are "woven into" the application using special compilers called *weavers*. Weaving can be done at all binding times such as system construction, generation time, and runtime. Join points where advice invocation code is woven in are called *join point shadows* [HH04].

Implementations of the AOP paradigm on top of the Java Virtual Machine (JVM) include AspectJ [Kic+01] (which has incorporated AspectWerkz [Vas04; Bon04] since 2005), CaesarJ [Ara+06], Spring AOP [Piv19], PROSE [NA05], and many more [Bri+05]. A compiler for the aspect-oriented language consists of a module for evaluating pointcuts and the aforementioned weaver, besides the elements of a traditional compiler. After evaluating a pointcut, the resulting join point shadows are forwarded to the weaver. At weaving time, it cannot be decided for all join point shadows whether pointcuts apply or not. That happens when the pointcut quantifies over dynamic properties such as `cflow`, `target`, `this`, and `args` pointcut designators.[5] Thus, for some join points,

[5]Pointcut designators for dynamic properties such as `cflow` quantify over the control flow, and `target`, `this`, and `args` can be used to filter objects from the join point by their dynamic type, and check whether they are the active or an argument object.

Listing 3.3 An AspectJ example aspect adding context-dependent behavior to a person. If asked for the number in the control flow of a business the business-related number is returned instead.

```
public aspect OfficeNumber {
  // inter-type declarations to enrich Person
  private PhoneNumber Person.officeNumber =
    new PhoneNumber("0049", "0351", "123456");
  public PhoneNumber Person.getOfficeNumber() {
    return officeNumber;
  }

  public pointcut askOfficeNumber(Person p) :
    // in the context of Business
    cflow(execution( * Business.*(..) ))
    // calling getNumber() on p
    && call( PhoneNumber Person.getNumber() ) && target(p);

  // replace getNumber() with getBusinessNumber()
  PhoneNumber around(Person p) : askOfficeNumber(p) {
    return p.getOfficeNumber();
  }
}
```

the advice invocation logic is compiled into the application and evaluated at runtime, called *residual* [Hau05].

An aspect-oriented implementation using AspectJ of the context-dependent behavior adaptation is shown in Listing 3.3. Inter-type declarations defined inside the aspect allow the addition of members to external classes. Therefore, the `Person` class could be extended with an office number and the behavior to return that number in the context of a business. To be able to describe that behavior we must define that whenever `Person.getNumber()` is called in the control flow of any functions defined in `Business` we must *intercept* the call and *delegate* to `Person.getOfficeNumber()`.

Capturing the semantics of context-dependent advice invocation in a pointcut definition is a verbose and error-prone task. Remember, how aspect-oriented compilers (partially) evaluate those pointcuts, and advice invocation is woven into the join points. As such, the aspect compiler produces a verbose description of aspects in an object-oriented programming language which incurs an overhead [HS07]. The reason is, that function invocations and member accesses are typical locations for join point shadows that will be decorated with advice invocation logic (i.e., residues). Since the object-oriented target machine model does not understand aspect semantics residues consist of multiple instructions instead of simple invocations. Standard object-oriented optimizations such as caching the result of the dynamic binding at the call site do not apply to the aspect-oriented execution semantics where advice code is being implemented externally to the advised class or object which obscures control flow. The performance penalty is not only observable when the aspect is woven and active, but also when no advice is executed due to advice-guarding predicates. Besides the overhead related to advice invocation, there is also the overhead of the pointcut evaluator as well as the time it takes to weave pointcut invocation into the program.

Moreover, the focus of Aspect-oriented Programming is on cross-cutting, class-centric aspects. Object-centric, so-called *instance-local* aspects, introduce even higher performance penalties. One reason for this is that they prevent inlining, which is one of the most effective optimisations [Hau05].

3.1.2 Context-Oriented Programming

While AOP is able to handle *homogenous* cross-cutting concerns, Context-oriented Programming (COP) [CH05; HCN08] is concerned with *heterogenous* cross-cutting concerns [ALS08]. To adapt the behavior of an application to well-known contexts COP applies pre-defined contextual variations. As shown in the prior section, to achieve the same with Aspect-oriented Programming, context-sensitive adaptations have to be embedded in the predicates of a pointcut. Context-oriented Programming provides dedicated language support to achieve this. Similar to AOP the base program is altered at join points, mostly with the granularity of methods. The components of a COP program are presented in Figure 3.2. To achieve contextual variation, *layers* encapsulate the context-dependent behavior implemented in *partial methods*. Partial methods can be executed before, after, or replace a base method. To apply variants from multiple layers, partial methods may use `proceed` as a mechanism to delegate to the next active layer. The adaptation is guided by the activation and deactivation of appropriate layers. That can be defined by imperatively annotating the program with scopes, in a declarative [AHL13] way, or a reactive programming style [App+10] driven by events. In COP, the rewrite process is called *sideway composition* [HMI13] as the inheritance hierarchy of the base program is enhanced orthogonally at runtime.

In an object-oriented execution model, the execution of a function can be understood as a two-dimensional dispatch taking the dynamic type of the receiver object and the

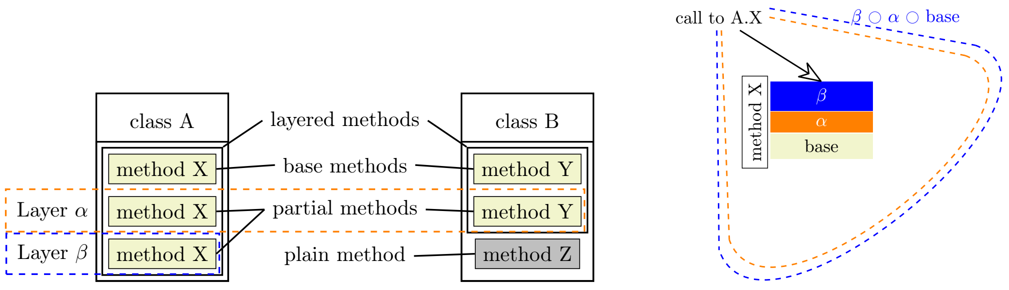


Figure 3.2: An overview of classes and layers in context-oriented programming and their influence on the execution semantics (adapted from [Lin+11, Fig.1]).

name of the function to be executed. In COP, dispatch is extended to four dimensions, as shown in Figure 2.1, which does not only take the receiver object and the name of the function but also the sender object and the context of the actual message sent into account [HCN08]. Indeed, the first implementation, called Flavors [Moo86], used the multiple-dispatch provided by Lisp to dispatch to modularized implementations of partial methods. In ContextL [CH05] the Common Lisp Object System (CLOS) [Bob+88] is extended with context-oriented language constructs without the need of the programmer to design the inheritance hierarchy to fit into multiple-dispatch carefully.

With *single dispatch* found in modern object-oriented programming languages such as C++, Smalltalk, and Java, dispatch only takes the runtime type of the receiver and the method name into account. Thus, dispatching requires a new dynamic binding mechanism to subscribe partial methods from activated layers to a call site and to dispatch them to the respective partial methods on invoking base methods. To realize this semantics the implementations rely on verbose descriptions of partial methods and layers resulting in a similar problem we have seen in AOP (see Section 3.1.1). In the following, we will bring our attention to the implementations of COP approaches based on single dispatch.

Implementations focused on using libraries [App+09; SGP12], for example, context-aware execution semantics implemented using imperative control flow in single dispatched languages like Self [UOK14], or as source-to-source compilers [AH12b]. However, there are also declarative approaches that allow to definition of layer composition rules resembling pointcuts of aspect-oriented programming languages [AH12a].

The following will exemplarily introduce JCOP [AH12b] a language extension of Java that provides control-flow-specific scoped activation of layers. Executing a function within the scope requires consulting the stack for active layers to delegate to the appropriate partial method. Listing 3.4 shows how to implement the office number use-case using layers that change the implementation of `Person::askNumber`. Layer composition allows specifying the sequence of layer activations and excluding layers from the composition. Every partial method can provide its context-dependent implementation and forward execution to the next active layer using `proceed`. Layers can form their inheritance hierarchy and overwritten implementations of partial methods can be delegated to with `superproceed`. JCOP also features declarative context-activation support, providing a simple pointcut model limited to functions only [AHL13]—highlighting its close relationship to aspect-oriented programming. Those pointcuts can be defined in a `contextclass` by either defining functions or predicates using `on` or `with` that enable layer activation. Listing 3.5 provides a short example of how to declaratively define layer activations.

COP language abstractions, namely layers and dynamic activation, increase the expressiveness of programming languages. However, these abstractions come with a cost. Across different implementations severe performance penalties exist [App+09]. It is of interest to note that COP combines context-dependent variations and the composition of crosscutting concerns at a class-wide granularity. Context-dependent variations with object-level granularity are not the primary feature of the language constructs and require manual work.

Listing 3.4 A JCop example with scoped layer activation.

```
public class Business {
  public PhoneNumber getNumber(Person p) {
    with(new OfficeNumberLayer()) {
      return p.askNumber()
    }
  }
}

public layer OfficeNumberLayer {
    public PhoneNumber p.Person.askNumber() {
        return new PhoneNumber("0049", "0351", "123456");
    }
}
```

Listing 3.5 A JCop example with declarative layer activation.

```
public contextclass OfficeNumber {
  on(PhoneNumber p.Person.askNumber()) : with(new OfficeNumberLayer());
}
```

3.2 The Role-oriented Paradigm

Turing Award winner Charles W. Bachman viewed the "*programmer as a navigator*" [Bac73] which became able to explore and navigate data records in a database instead of the data coming from a tape. This change in the possibilities to store, access, and model data records allows a completely new approach to data modeling using the concepts of items and sets. The first account for roles in modeling can be found by Bachman and Daya; they proposed the *role data model* as an extension to the network model [BD77]. It describes roles as "*a defined behavior pattern which may be assumed by entities of different kinds.*" [BD77, p.2]. The role data model defines the role types that can be assumed by different entity types coherently. Defining a data model for an object-oriented database can be seen as a similar activity to defining the class model during conceptual modeling in object-oriented analysis. Therefore it is not surprising that Reenskaug et al. proposed a conceptual framework as a result of 20 years of consulting experience, named OOram [RWL96] which identifies and uses roles. Moreover, they state:

> "All information that can be expressed in a class-based model can be expressed in a role model. All information that can be expressed in an object-based model can be expressed in the same role model."
>
> – Reenskaug et al. [RWL96]

They propose that static properties and relations, such as attributes and inheritance relations, are best expressed in terms of classes. Dynamic properties, such as use cases and scenarios, are best expressed in terms of objects. The role model is a combined model that can express both dimensions.

In the literature the term "*role*" occurred in different contexts such as Role Based Access Control (RBAC) [FCK+95], knowledge representation, conceptual modeling, data modeling, and object-oriented design and implementation [Ste00a]. While the application of roles in data modeling and conceptual modeling has proven to be successful it had next to no impact on software programming languages and development in practice.[6] However, role modeling had a prevailing impact on the programming language research community inspired by the idea of a flexible model that can represent static and dynamic properties of entities. Multiple role-based programming languages have been proposed that extend the concept of OOP to include roles. The approaches range from static implementations where role-playing can never be changed, to semi-dynamic where role-playing is allowed in a pre-defined manner, to fully dynamic with no prior restrictions. We review role-based programming languages and account for the classification of their features in Chapter 5.

Since early 2000 computer scientists encountered a new research problem. The software has been deployed on mobile devices which became ubiquitously available and required "*the necessity to integrate varying functionality at runtime being activated depending on the current context.*" [Pie+12]. In a broader sense, Piechnick et al. argued that distributed mobile applications need to continuously adapt to different contexts to fulfill their requirements. This context-aware software adaptation led to the introduction of notions of *contexts* or *compartments* as first-class citizens to role-based modeling and programming languages [Küh+14], leveraging the potential of roles about the ease of extensibility and (unanticipated) adaptation of existing program code.

The roles an object plays modify its behavior as long as the context that provides the roles is active. For example, the context is the `Business` where a `Person` plays the role of an `Employee`. Figure 3.1 shows a Compartment Role Object Model (CROM) that models the example mentioned above. In that model, `Business` is modeled as a compartment. `Employee` is a role that is played in the context of a business and played by `Person`. That means any `Employee` role is dedicated to a `Person` object while also being dependent on the `Business` object itself. This type of dependency forms dependent types, leading to *family polymorphism* [Ern01; Her04]. While in recent years the context-dependent nature of roles again became of interest to the research community there is no combined concept for both relational and context-dependent approaches [Küh+15].

3.2.1 Role-based Modeling and Programming Languages

The successful adoption of roles in software analysis and modeling [AR92; RWL96; RG98] led to a demand for programming language support. Because software inherently becomes more complex, it subsequently becomes more complicated to evolve or adapt the software. Role-oriented Programming (ROP) is able to increase (1) com-

[6]Mads Torgersen opened a discussion on Dec 1, 2021, called "*Roles and extensions in C#*" [Tor21] that proposes language and compiler extensions to provide some of the proposed role features of [Küh+14] to C#.

prehensibility as the system is represented with less complex components with a clear boundary, (2) changeability and adaptability because changes are isolated inside individual components, and (3) reusability because components can be plugged and played. However, during the last forty years, different views of roles emerged. An attempt to reconcile and classify existing approaches by their features ascribed to roles has been made [Ste00a; Küh+14]. Steimann [Ste00b] provided a list of features for *relational* roles. In Kühn et al. [Küh+14] the list of features is further extended to account for *context-dependent* roles and a taxonomy for the models and meta-models of role-oriented modeling and programming languages is proposed. They highlight that some features are mutually exclusive and propose a feature model and intra- and inter-feature constraints to generate meta-models from feature configurations. Each feature configuration can be represented in Compartment Role Object Model (CROM), a metamodel representing a particular role-oriented programming or modeling language that can represent the chosen features. Compartment Role Object Instance Model (CROI), a metamodel to represent instances concerning a CROM model, can be leveraged to check whether instances conform to a CROM model [Küh+15]. However, as a consequence of the static nature of the model, CROM cannot capture the dynamic semantics of these languages, while CROI can check a specific snapshot only.

The first approaches to support roles in programming languages resulted in the role concept being hidden in the implementation of the host language [Bäu+97; Fow97], effectively losing the notion of roles (and contexts) when implementing the role models. Bäumer et al. have introduced the Role Object Pattern (ROP) because of the need for a flexible design pattern that allowed for unanticipated changes without the need to recompile the whole application [Bäu+97]. The pattern represents players and roles in an inheritance hierarchy. An abstract class of the entity represents the root, the player is split into a *core* class that implements the basic behavior. Role classes are inherited from the root and delegated to their core. It allows modeling of different views of an object designed as role objects which are dynamically added and removed from the core object. The pattern has no realization for context-dependent roles. Furthermore, possible implementation techniques heavily depend on the implementation language. For example, Andersen et al. stated that "*... to realize the joint implementation of a composite role [...] can often be constructed by multiply inheriting the classes of the synthesized roles.*" [AR92] If the language of the implementation does not support multiple inheritance, nontrivial restructuring must be carried out. Fowler suggested different implementation strategies for roles but highlighted that these strategies are only to fulfill functional requirements not taking non-functional requirements such as performance into account [Fow97].

Role-oriented programming distinguishes between the base entities themselves and the roles they play in a collaboration. This provides explicit support for object collaboration in a way not normally supported by language features [HHG90]. At runtime, the collaboration among objects is predetermined by the functional requirements of the application. This means that at different points in time, different parts of an object's interface are used by other objects. This context-based usage cannot be directly represented in OOP but is scattered over the object-oriented program. The reason is that functions are reused from multiple concerns or may encapsulate code for multiple different concerns. Previously discussed approaches achieve Separation of Concerns at the granularity of functions, classes, or modules. Roles take the idea further, as "*no object is an island. All objects stand in relationship to others*" [BC89].

The implementation techniques used for role-oriented programming can be assigned to two groups naturally following their target languages' division into *class-based* and *prototype-based* (or object-based) OOP [Bor86]. In class-based object-oriented programming languages, classes include all implementations for the foreseen collaborations of their instances, resulting in functions and interfaces that contribute to multiple collaborations. On the other hand, instead of defining classes with interfaces that span all use cases, the dynamic composition of collaborating objects should define the interface. While the former represents the traditional object-oriented approach to design and implement software the latter can integrate software into *use contexts* that have not been foreseen by their designers [RG98].

3.2.2 Roles in Prototype-based Object-oriented Languages

Object-oriented programming that follows the prototype-based or object-based concept is concerned with individual objects and their relations to other objects. A well-known representative of this category is JavaScript (JS), a language heavily inspired by Self [CU89; CUL89]. In this category, objects consist of *named slots*, each of which contains a reference to another object. Some slots may be *parent* slots, others can be a reference to methods. To create a new object an existing object — the *prototype* — is simply *cloned* (shallow-copied). Because these languages are dynamically typed, they support objects admitting or removing properties.

Method dispatch in this category works in the way that the receiving object's slots are searched for a slot name equal to the name of the method. However, an optimization commonly used nowadays is to create a Polymorphic Inline Cache (PIC) [HCU91] that stores the prototype an object is constructed from.

The rest of this section, introduces the Scala ROles Language (SCROLL) [LA15; Leu17b] a role-oriented programming language providing contextual roles. SCROLL, following the idea of CROM, provides a MOP for compartments, roles, and base

Listing 3.6 The Scala ROles Language implementation of adding context-dependent behavior to a person. If asked for the number in the scope of a business the business-related number is returned instead.

```scala
object BusinessNumberExample {
  case class Person(privateNumber: PhoneNumber)

  class Business extends Compartment {
    class Employee(officeNumber: PhoneNumber) {
      def getNumber(): PhoneNumber = officeNumber
    }
  }
  def main(args: Array[String]): Unit = {
    val joe = Person(new PhoneNumber("0049", "0351", "987654"))
    val business = new Business {
      joe play new Employee(new PhoneNumber("0049", "0351", "123456"))
      +joe.getNumber // Returns Employee::getNumber
    }
  }
}
```

types implemented as an internal DSL in Scala. While Scala is a class-based, functional, object-oriented programming language, SCROLL uses a combination of compiler rewrites, implicit arguments, and the ease of embedding domain-specific elements in the host language Scala. For this reason, we use SCROLL as a representative of a prototype-based object-oriented implementation for contextual roles. Instead of being viewed as a programming language on its own, it could also be viewed as an internal DSL in a class-based object-oriented programming language.

In Listing 3.6 we provided an implementation of the running example in SCROLL. The class `Business` must extend the `Compartment` class of the provided MOP to become treated as a context. The role `Employee` is defined as an internal class of the compartment. SCROLL chose not to provide annotations or metafunctions to constrain the *plays* relation. Any class can fill the role. A role that defines methods with the same name and signature as the player can replace methods. A method that is neither implemented in one of the roles that will be played when the method is called nor defined by the player's type will result in a runtime error. SCROLL can enforce this property only at runtime, which corresponds to the prototype-based object-oriented implemen-

Listing 3.7 SCROLL uses a combination of compiler rewrites, implicit conversion of arguments, and the ease of embedding domain-specific elements in the host language Scala.

```
+joe.getNumber
        ~~> joe.unary_+().getNumber()
        ~~> new Player[Person](joe).getNumber()
        ~~> new Player[Person](joe).applyDynamic("getNumber")()
```

tation of the approach. In line with Self's `undefinedSelector` error, SCROLL throws an `DynamicBehaviorNotFound` exception when a method cannot be resolved.

In his thesis Leuthäuser describes that "*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.*" [Leu17a, p. 39]. As such, SCROLL hooks into method dispatch using a combination of compiler rewrites, implicit arguments, and the ease of embedding domain-specific elements in the host language Scala using *dynamic traits*. The statement `+joe.getNumber` defines the role invocation on the object `joe` represented by the unary plus operator. In Listing 3.7 we exercised the automatic rewrite the Scala compiler will apply to statements using this operator. It shows on one side how the DSL can be nicely embedded into Scala code and on the other side how seemingly normal type-safe function calls will be rewritten to type-unsafe operations that may fail at runtime.

Languages such as SCROLL keep a representation of the state of the application at runtime. The data structure is coined the *Role-Play-Graph*.

3.2.3 Roles in Class-based Object-oriented Languages

To organize object-oriented programs there are two key principles: One is to identify and categorize similar objects into classes known as classification. The other is how relationships of classes are organized into a class hierarchy using generalization, i.e., sharing common behavior and attributes in a superclass. Classes encapsulate the implementation and provide interfaces. OOP excels at representing the structure of a domain [Bor86] but struggles to represent dynamic collaborations of objects over time. In most class-based object-oriented systems the association between an instance of a class and the class itself is permanent [GSR96].

Most object-oriented programming languages implement roles and compartments as separate classes. Role-playing adds another (spatial) dimension to the principle of generalization in OOP. It introduces a relationship between classes or their instances that is orthogonal to inheritance and may be valid for a limited time. This relationship is, in most approaches, not implemented via inheritance but emulated with delegation [Her10].

In the following, we present ObjectTeams (OT) [Her03] and ObjectTeams/Java (OTJ) [Her05; Her07], the implementation of the model in Java.[7] ObjectTeams is a representative of the class-based object-oriented approaches. The language bridges the statically typed, class-based object-oriented world and the dynamic, object-based world interestingly. OT introduces aspect-oriented and role-based concepts that are smoothly integrated with object-oriented concepts like inheritance and polymorphism. Highlighting the possible crosscutting nature of object collaboration the semantic of roles in ObjectTeams (OT) is similar to instance-local dynamic aspects. The context in which roles can be played is represented using *teams*; a higher-order module for contained *roles*. The roles themselves are contained as inner classes within a team. In contrast to languages presented in the prototype-based approach, ObjectTeams is a type-safe programming language. Types that are valid to fill a role can be declared by the `playedBy` relation restricting role-playing to the defined base type. Where it *gaps* the class-based and object-based world is when taking a look at *bindings*. A *callin* method binding intercepts the control flow at a method of the base entity and redirects it to a role method. That is how an object can delegate a method call to another object to handle it. As OT introduces aspect-oriented features these bindings can come in different flavors such as before or after the specified method. When specifying a *replace* callin binding, the effect is the same as overriding a method in the context of inheritance. At runtime, there exist several means to define whether a binding is effective, i.e., whether or not the interception takes place. For example, a team instance can be *activated* or deactivated, which has the effect that all callin bindings of all contained roles are enabled or disabled.

Listing 3.8 presents an implementation of the role model shown in Figure 3.1. OTJ is an extension to the Java programming language. The OTJ compiler provides static semantic analysis such as type-checking the extended Java program, the playedBy relations, and method bindings. The example uses the concept of *declared lifting* where a specific `Person` class is lifted to its `Employee` role using `Person as Employee`. Methods that define new behavior affecting the base class are prefixed with `callin`. For each `callin` there must be a binding declaration. In our example, it declares that `Person::getNumber` is *replaced* by `Employee::getNumber`.

[7]While the model is named ObjectTeams (OT), the implementation of the model in Java is denoted ObjectTeams/Java (OTJ). We will use ObjectTeams (OT) throughout the thesis for both model and implementation. We will not discuss prior iterations of ObjectTeams [HM00; MSL01; HM01; Vei14].

Listing 3.8 ObjectTeams Java Example

```
public team class Business {
  public void employ(Person as Employee emp) { } // Declared lifting
  public class Employee playedBy Person // Adapt iff Person has role
    base when(Business.this.hasRole(base, Employee.class)) {
    private PhoneNumber officeNumber;
    callin PhoneNumber getNumber() {
      return officeNumber;
    }
    // Replace Person::getNumber with Employee::getNumber
    PhoneNumber getNumber() <- replace PhoneNumber getNumber();
  }
}
Business business = new Business();
Person joe = new Person();
business.activate();
business.employ(joe);
joe.getNumber(); // Returns Employee::getNumber
```

4 Foundations of Partial Evaluation

"Computers are incredibly fast, accurate, and stupid. Human beings are incredibly slow, inaccurate, and brilliant. Together they are powerful beyond imagination."

— Albert Einstein

This chapter provides a brief introduction to the basic mechanisms of Partial Evaluation. We will show how introducing different compilation stages allows to partially evaluate a program and how that may be applied to program generation and program optimization. Last, we introduce how the reapplication of Partial Evaluation is program generation and produces compilers out of interpreters.

4.1 Partial Evaluation

Partial Evaluation (PE) provides a unifying paradigm that represents tasks commonly seen in program optimization, compilation, interpretation, and the generation of automatic program generators [Jon96]. It is a technique to *specialize* a (general) program with regard to a given input. The main motivation to apply PE is speed, given that an unspecialized program has to execute more code than the equivalent specialized program.

Assume a fixed set of data values including program terms, for example, first-order functional programs or Lisp's `list` [JGS94; Jon96]. Given a program implemented in the discussed language S, i.e., $program_S$, then $[\![program_S]\!]$ denotes its meaning. The program meaning function $[\![\_]\!]$ (or interpretation) is described equationally as:

$$[\![program_S]\!] in_1, \ldots, in_n = output$$

Thus, *output* is the result of interpreting $program_S$ with the input values $[in_1, \ldots, in_n]$ given the program terminates. Assume we know any value in_k with $k \in \{1, \ldots, n\}$ then the program can be partially evaluated given the value in_k producing an equivalent *specialized* program $program'_S$ where the occurrence of the variable has been replaced by the value in_k and possible computations or optimizations, such as constant folding, had been carried out.

$$[\![program_S]\!] in_1, \ldots, in_n \equiv [\![program'_S]\!] in_1, \ldots, in_{k-1}, in_{k+1}, \ldots, in_n = output$$

We call input values that are known at compile-time *static* values while the input values only known until execution are called *dynamic* values. Given the notion of *stages*, one could say that a static value is known at the *evaluation stage* while the dynamic values must be retained as variables in the target program. Static values applied in the first stage are written in brackets [_] while dynamic values applied in the second stage are without brackets.

While compilers often embed Partial Evaluation directly in program optimization it can also be treated as a separate component. In that sense, the partial evaluator is another program called *mix* that evaluates a program and its static inputs and returns the specialized program w.r.t. to those inputs.

$$\begin{aligned}
&[\![[\![mix]\!][program_S, in_k]]\!] in_1, \ldots, in_{k-1}, in_{k+1}, \ldots, in_n \\
&\qquad \equiv [\![program'_S]\!] in_1, \ldots, in_{k-1}, in_{k+1}, \ldots, in_n \\
&\qquad = output
\end{aligned}$$

It has to be highlighted that PE requires that there is a pre-phase and a post-phase (that is to check terms) and that, given the values of terms in the pre-phase, a reduction is possible. In [DP01] it is shown that modal logic could be used to check whether a separation of phases is possible. Otherwise, *annotations* tell whether a variable is eliminable or residual [Jon96]. Program generators and compiler-compilers are well-known from the domain of parser generators and have been applied, for example, to produce specialized LR(k) parsers [ST95]. In this work, however, we will not look at compiler-compiler generators that specialize from language specification languages, but compilers that use PE to specialize predefined programming languages.

4.2 Futamura Projections

Traditionally we use compilers to transform a high-level source language S to a lower-level target language L (or executable). But instead of a compiler, the gap between the source language S and the lower-level target language L may also be bridged with the use of an interpreter. The interpreter interprets statements in language S, evaluates static inputs, and produces values or statements in language L. Given the notion that we just introduced in the prior section, we may describe the interpretation pattern equationally as:

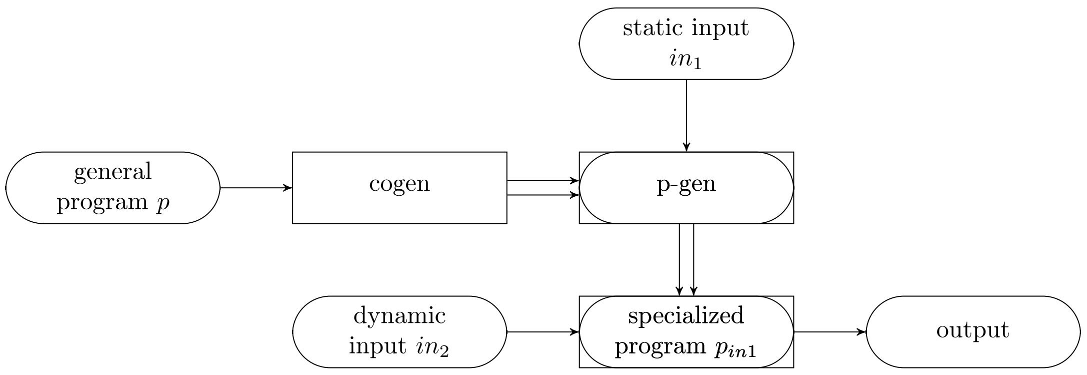


Figure 4.1: `cogen` is a generator of program generators (inspired by [Jon96, Fig.8]) realized by self-application. Rounded boxes represent data, rectangles represent generators, a combination means both. Arrows represent data flow.

$$[\![[\![interpreter_S^L]\!] [program_S]]\!] input = [\![program_L]\!] input = output$$

At some point the interpretation overhead of the *program generator* to construct $program_L$ may be unacceptable. In that case one may want to produce faster generators of specialized programs. An overview of such a generator of program generators is given in Figure 4.1. The generator of program generators `cogen` accepts a two-input program p and generates the program generator `p-gen`. The program generator or *generating extension* as coined by Ershov [Ers82] produces a specialized program p_{in1} given the known value in_1 for the first input of p. Program p_{in1} produces the same output given the dynamic input in_2 as p would compute given the input $in1$ and $in2$.

We encourage the interested reader to visit [WP17] where a diagrammatic approach to Futamura Projections is taken.

4.2.1 Compiling by First Futamura Projection

Interestingly, the interpreter can also be partially evaluated with the program being its static input. Thus, *mix* can be used to compile. The resulting target $program_L$ is a *specialized interpreter* of the given program. This is often called the 1st *Futamura projection* which has been first reported in [Fut71].

$$\begin{aligned}
output &= [\![program_S]\!] input \\
&\equiv [\![interpreter^L_S]\!] [program_S, input] \\
&\equiv [\![[\![mix]\!] [interpreter^L_S, program_S]]\!] input \\
&\equiv [\![program_L]\!] input
\end{aligned}$$

4.2.2 Compiler Generation by Second Futamura Projection

With our partial evaluator *mix* we can also generate a compiler which can be used standalone. Let us highlight the difference between the first and second futamura projection. In the first futamura projection, *mix* is evaluated on the interpreter with the program as its input to directly generate the target program. In the second futamura projection, *mix* is applied to itself with the interpreter as static input to generate a specialized version of the interpreter.

$$compiler^L_S = [\![mix]\!] [mix, interpreter^L_S]$$

We can now use the compiler to compile input programs written in language S to programs in language L. A concrete example can be found in [JGS93, Ch.4].

$$program_L = [\![compiler^L_S]\!] program_S$$

4.2.3 Compiler-Compiler Generation by Third Futamura Projection

The compiler generator `cogen` is a program that transforms interpreters into compilers. It generates compilers that are versions of *mix* itself specialized to various interpreters it is applied to. Let us show how `cogen` can be created given the scenario from Figure 4.1.

$$\begin{aligned}
&[\![program]\!] [in_1, in_2] \\
&\qquad \equiv [\![[\![mix]\!] [program, in_1]]\!] in_2 \\
&\qquad \equiv [\![[\![[\![mix]\!] [mix, program]]\!] in_1]\!] in_2 \\
&\qquad \equiv [\![[\![[\![[\![mix]\!] [mix, mix]]\!] program]\!] in_1]\!] in_2 \\
&\qquad \equiv [\![[\![cogen]\!] [program, in_1]]\!] in_2
\end{aligned}$$

While being hard to understand intuitively from the above set of equivalences we may imagine *program* is a program parser and in_1 a given grammar as its known input.

That means that `cogen` transforms the input grammar into a specialized parser program for that grammar. This had been realized to produce specialized LR(k) parsers [ST95].

Part II

Problem Analysis

5 Implementation Techniques for Role-oriented Programming Languages

"The language designer should be familiar with many alternative features designed by others, and should have excellent judgement in choosing the best and rejecting any that are mutually inconsistent... One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation."

— Tony Hoare

Almost half a century ago, the concept of roles was introduced, which led to the development of numerous approaches. In this chapter, we provide an overview of previous literature reviews that have explored role-oriented programming and modeling languages. We will highlight their shortcomings concerning the reported information on the *implementation techniques* used. To fill this gap, we present a review of role-oriented programming languages published between 1990 and 2020 designed with a strong focus on the execution model of Role-oriented Programming. We are interested in the mapping of language constructs used to represent roles to the target execution environments, the representations of the *plays* relation, and the implementation of method dispatches of role-playing objects in the target execution environments. To previously unreviewed literature, we apply a classification of role-oriented programming and modeling languages found in the literature. Last, we extend a classification found in the literature to contextual roles and apply the extended classification.

5.1 Features of Role-oriented Programming and Modeling Languages

In the past, multiple reviews have been conducted focusing on different aspects of role-oriented programming and modeling languages. In the following, we will introduce them in chronological order by date of publication.

Kniesel presented a taxonomy concerning the dynamic binding, the support of multiple perspectives, and the interface of references to role-playing objects [Kni96]. The authors reviewed role-oriented programming languages published up to 1996 and grouped reviewed approaches into distinct groups of *roles with objects* and *objects with roles*. In the former objects may acquire new roles but the new behavior is not perceived through existing references. This class comprises roles as external perspectives distinct from the role-playing object such as aspects or views. In the latter approach, existing references to objects reflect changes such that the interface changes and old behavior is adapted or new behavior becomes available. The review highlights interesting points discussing the implications of accessing object attributes over time and the perceived changes applied to the object.

The review of Steimann includes the results of prior conducted reviews on the topic of role-oriented modeling and programming languages. The authors compiled a list of 15 features ascribed to role languages [Ste00b] which can be found at the top part of Table 5.1. Steimann is concerned about the potential of roles for conceptual modeling of software systems. In their view, the implementation of roles can be translated to role interfaces which will be implemented by role-playing classes [Ste01]. A role change is a mere cast of an object to one of the role's interfaces. While the review captured features attributed to role-oriented modeling languages it does not distinguish between the meta-levels. We think that it does not sufficiently consider the many nuances programming languages offer to host the presented features.

Kühn et al. extended the list of features proposed by Steimann with features concerning contextual roles [Küh+14]. We list the set of proposed features in Table 5.1. Additionally, the authors reviewed role languages published between 2000 and 2015. They highlight that some features proposed by Steimann are mutually exclusive and propose a feature model and inter-feature constraints to generate valid feature configurations. Each feature configuration can be represented in Compartment Role Object Model (CROM), a metamodel representing the provided features of a particular role-oriented programming or modeling language [Küh+14]. However, due to the focus on modeling languages and as a consequence of the static nature of the model, it cannot be used to reason about the semantics of role-oriented programming languages. It neglects the rich set of semantics on method lookup found in the literature making it only applicable to a small subset of approaches. CROM does capture the static semantics and can be used to generate skeleton implementations for role-oriented programs. This has been exercised for the code generation of Scala ROles Language from CROM models [Leu19]. This is in part possible because SCROLL closely mimics the semantics of CROM realizing many of the constraints provided by the meta-model [Leu17a, Sec. 9.2.6]. This allows to generate code that uses the program logic of SCROLL to check constraints for role-playing and cardinalities, to name a view properties. While CROI can check the well-formedness of a model instance [Küh+15] it may not be applicable to represent and check an (abstract) program as it does not capture any dynamic semantics usually found to describe dynamic programming language semantics.

A taxonomy and feature model to describe aspects of the semantics of role-oriented programming languages published until 2006 was proposed by Graversen [Gra06]. To instantiate configurations of the feature model they opted for a framework approach because "*using a dynamically typed language [Python [Van+07]] helped to avoid all the problems raised [...] on types*" [Gra06, p.148]. Thus, instead of providing a grammar plus lexer and parser, a configuration can be elaborated as a Python program using (meta-) classes and (meta-) functions provided by the framework. Consequently, the review does not discuss static semantics to map the terms of a program to a feature configuration. An interesting line of thought is presented in [Gra06, p.207] on role termination that resembles similarity to problems arising from dangling pointers in use-after-free, that is a pointer or reference to an object that was previously freed. While caching and reusing roles in the process of *role lifting* (cf. Object Teams in Chapter 3.2.3) efficiently speeds up role-oriented programs and reduces the required memory of an application the question is when to remove a role. A dedicated role termination mechanism unlinks a role instance from the player, however, it is unspecified what happens if the role is used after being unregistered. We imply that all *roles with objects* kind of approaches that externalize roles, i.e., allow references to roles, suffer from the same problem.

5.2 A Review on Role-oriented Programming Languages

Role-oriented programming has been exercised since its inception as a data model for object-oriented databases in 1977 [BD77]. The aforementioned reviews unearthed a trove of literature about the features of roles and role-oriented modeling and programming languages and their applicability to model or program software systems. However, none of the aforementioned reviews emphasized the *implementation techniques* used to realize the provided language features. We think that the way particular features are realized has a huge impact on the perceived performance of a programming language. For example, there exist multiple runtimes for the Ruby [FM08] programming language that each deliver different run-time performance [Sea16]. The reason is the different design decisions, different techniques, and different programming languages used in the runtimes. This applies to each mainstream programming language that has multiple different implementations.

Table 5.1: 26 classifying features extracted from [Küh+14]. Features 1-15 have been proposed by Steimann [Ste00b]. Features 16 - 26 have been proposed from Kühn et al. [Küh+14].

#	Description	Meta Level
1.	Roles have properties and behaviors	(M1, M0)
2.	Roles depend on relationships	(M1)
3.	Objects may play different roles simultaneously	(M1, M0)
4.	Objects may play the same role (type) several times	(M0)
5.	Objects may acquire and abandon roles dynamically	(M0)
6.	The sequence of role acquisition and removal may be restricted	(M1, M0)
7.	Unrelated objects can play the same role	(M1)
8.	Roles can play roles	(M1, M0)
9.	Roles can be transferred between objects	(M0)
10.	The state of an object can be role-specific	(M0)
11.	Features of an object can be role-specific	(M1)
12.	Roles restrict access	(M0)
13.	Different roles may share structure and behavior	(M1)
14.	An object and its roles share identity	(M0)
15.	An object and its roles have different identities	(M0)
16.	Relationships between roles can be constrained	(M1)
17.	There may be constraints between relationships	(M1)
18.	Roles can be grouped and constrained together	(M1)
19.	Roles depend on compartments	(M1, M0)
20.	Compartments have properties and behaviors	(M1, M0)
21.	A role can be part of several compartments	(M1, M0)
22.	Compartments may play roles like objects	(M1, M0)
23.	Compartments may play roles which are part of themselves	(M1, M0)
24.	Compartments can contain other compartments	(M1, M0)
25.	Different compartments may share structure and behavior	(M1)
26.	Compartments have their own identity	(M0)

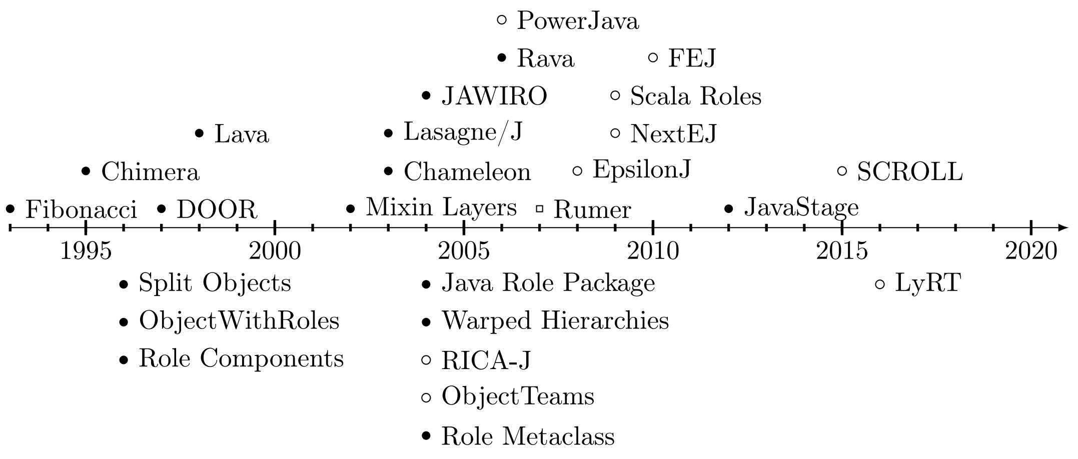


Figure 5.1: A timeline of publications of role-oriented languages which are attributed to be either behavioral (●), relational (□), or contextual (○) approaches.

5.2.1 Presentation Framework

To fill this gap, we present a review of role-oriented programming languages published between 1990 and 2020 designed with a strong focus on the features of Role-oriented Programming and, if possible, the implementation techniques used to realize these features. Starting from the aforementioned literature reviews [Kni96; Gra06; Ste00b; Küh+14] we did a forward and backward search on the referenced papers. In Figure 5.1 we show a timeline of the role-oriented programming languages that we identified in our review process. The review comprises a total of 25 role-oriented programming languages. We assigned 15 approaches to belong to the behavioral category, 1 approach to belong to the relational category, and 9 approaches to belong to the category of contextual role-oriented languages.

Our framework to analyze the ROP implementations focuses on execution models. We captured the implementation techniques used to support particular language features in the chosen target languages. It is especially important to understand *how* the object-oriented concept was extended considering the available language features, for example, multiple inheritance or multi-methods. We paid special attention to the mapping of the semantics of method lookup and method dispatch. The presentation framework presents selected topics of interest that help to understand, compare, and classify the different approaches.

Language Presentation The ROP language implementation is briefly described. We do not deliver a throughout description of the language features but consider the points important to understand the approach. Since ROP implementations support different features and provide solutions for different problems, we introduce the application scenario or use case where the approach is applied.

Architecture The overall architecture concerning the representation of roles in the ROP implementation is considered. This includes a brief overview of the formal or informal type system. We also present how the implementation considered access to ROP entities such as contexts and classes.

Programming Model Implementation This section of the framework is concerned with the representation of role-oriented entities in the target implementation as well as semantic properties concerning method lookup and dispatch.

Weaving or Mapping Implementation This section addresses concerns about the strategy of how application code is altered or how the role-oriented entities are mapped into the target runtime model.

Context and Role Instance Management This section of the framework considers how the ROP implementation associates instances with the objects or class for which they represent a role and behavior.

5.2.2 Classification of Role-oriented Programming Language Implementations

Table 5.2 classifies technical aspects such as the year of publication, the target programming language, and whether the type system has been formalized. We believe that the programming language used or targeted has a strong influence on the implementation techniques used. If the ROP implementation uses a custom syntax or extends the syntax of an existing programming language then a mapping to either a specific target language must be implemented or a custom language runtime must be designed. For this case we distinguish between a compiler—that is a "full-fledged" tool that *transforms* an input language into another language with a non-trivial mapping from one to the other—and a transpiler that *converts* language mechanisms between an input language into an output language. In other cases, the ROP implementation has to resort to the mechanisms provided by the programming language. For example, programming languages that provide meta-programming capabilities are better suited for internal DSLs and Metaobject Protocols than languages that do not. The difference between a MOP and an *internal DSL* is small. We specify that an approach uses a MOP if it defines metaclasses that must be inherited to enable the approach's mechanism. An internal DSL on the other hand can use the target programming language mechanisms to hide the process of creating and managing roles and contexts. The last resort is to base the implementation on a library or framework approach where the ROP implementation is managed by using library functions. In that sense, it is crucial to classify how approaches make roles and contexts available as language constructs and by what mechanism the method lookup is realized. Last we summarize how the *plays* relationship is implemented.

To give a detailed account from multiple perspectives we also classified each approach using the taxonomy of Kühn et al. [Küh+14]. The features proposed by Kühn et al. [Küh+14] are focused on "*the representation of roles and role models in role-based languages, [...] focused on the type level representation of roles rather than their implementation.*" [Küh17, p.32] It thus complements our implementation-focused review with an account for the structural properties of ROP implementations. The result of the evaluation is summarized in Table 5.3. For completeness, we included some role-oriented programming languages already classified in the literature [Küh+14; Leu17b; Ste00b] highlighted by a gray background. The rest of the section will present the ROP implementations using an extended classification scheme from [Kni96].

5.2.3 Behavioral Role-Oriented Programming Languages

5.2.3.1 Fibonacci

Fibonacci [AGO93; AGO95] is an object-oriented database programming language and successor of Galileo [ACO85; AGO88] which is a statically and strongly typed programming language.

Language Presentation The overall idea was motivated because "*techniques used by object-oriented languages for modeling objects and associations between sets of objects are not satisfactory*" [AGO95, p.1f]. As a database programming language, it builds an incarnation of the data model proposed by Bachmann [BD77]. It integrates an abstraction mechanism of a data model within the programming language to overcome the limitations of traditional approaches using ad hoc mechanisms to exchange information between a program and the Database Management System (DBMS). They observed that the "*object-oriented data model is best suited to represent complex data types, es-*

Table 5.2: An overview of the technical aspects of the implementations of Role-oriented Programming Languages and key principles applied to method lookup and dispatch.

Language	Year	Target	Type System	Approach	Method Lookup	Implementation
Behavioural						
Fibonacci	1993	PHAM	Informal	Compiler	Upwards/Double	Internal DAG
Chimera	1995	DSL	Informal	Compiler	Multiple Dispatch	Internal DAG
ObjectWithRoles[§]	1996	Smalltalk	Hidden	Meta Objects	Delegation	Internal DAG
Split Objects	1996	Self	Informal	Meta Objects	Upwards	Internal Tree
Role Components[§]	1996	C++	Hidden	Internal DSL	Upwards	Static Mixin
DOOR	1997	Scheme	Formal	Meta Objects	Delegation	Internal DAG
Darwin/Lava	1998	Java	Informal	Compiler	Consultation/Delegation	Inheritance
Mixin Layers	2002	Java	Hidden	Compiler	Upwards	Static Mixin
Chameleon	2003	Java	Hidden	Transpiler	Delegation	Inheritance
Lasagne/J	2003	Java	Hidden	Meta Objects	Delegation	Dynamic Proxy
Warped Hierarchies	2004	Self	Hidden	Library	Delegation	Multiple Inheritance
Java Role Package	2004	Java	Hidden	Meta Objects	Upwards Lookup	Internal DAG
JAWIRO	2004	Java	Hidden	Library	Consultation	Internal DAG
Role Metaclass[§]	2004	VODAK	Hidden	Meta Objects	Upwards Lookup	Internal DAG
Rava	2006	Java	Hidden	Transpiler	Delegation	Mediator/ROP
JavaStage	2012	Java	Hidden	Transpiler	Upwards	Static Mixin
Relational						
Rumer	2007	(Java)	Formal	(Transpiler)	–	Relational Algebra
Contextual						
ObjectTeams/Java	2004[†]	Bytecode	Informal	Compiler/Meta Objects	Delegation	Parallel Hierarchies
RICA-J	2004	Java	Hidden	Meta Objects	Delegation	Split Objects
PowerJava	2006	Java	Hidden	Transpiler	Delegation	Interface
EpsilonJ	2008	Java	Informal	Transpiler	Delegation	Split Objects
Scala Roles	2009	Scala	Hidden	Library	Delegation	Internal DAG
NextEJ[‡]	2009	(Java)	Informal	–	Upwards	Dynamic Mixin
Featherweight EJ[‡]	2010	Core Calculus	Formal	–	Upwards	Dynamic Mixin
SCROLL	2015	Scala	Hidden	Meta Objects	Delegation	Internal DAG
LyRT	2016	Java	Hidden	Library	Delegation	Internal Tree

[†] ObjectTeams provides two different compilers where the second, more flexible was published in 2010.
[‡] The language has never been implemented in a real system.
[§] Approaches without names coined by their authors have been given a name by us.

Table 5.3: Classification of role-based programming languages. An overview of the features is presented in Table 5.1. We differentiate between fully supported (■), partially supported (⊞), not supported (□), and not applicable (∅). Columns with a gray background have already been reported in literature [Küh+14; Leu17b; Tai17].

Feature	Fibonacci [AGO93; AGO95]	Chimera [CM95; BG95]	ObjectWithRoles [GSR96]	Split Objects [BD96; BD95]	Role Components [VN96a; VN96c; VN96b]	DOOR [WCL97a; WC98; WCL97b]	Darwin/Lava [Kni96; Kni99; CKC99]	Mixin Layers [SB02]	Chameleon [Gra03a; GØ03]	Lasagne/J [Tru+01; JT03]	Warped Hierarchies [VDD04]	Java Role Package [ST04]	JAWIRO [SE04; SE06]	Role Metaclass [DPZ02; DPZ04]	Object Teams [Her05; Her07]	RICA-J [SO04]	Rava [Che+06]	PowerJava [BBvdT06a; BBvdT06b]	Rumer [BGE07; Bal11]	EpsilonJ [MT08]	Scala Roles [Pra08; PO08]	NextEJ [KT09]	JavaStage [BA12]	SCROLL [LA15]	LyRT [Tai+16b; Tai+16a]
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	□	∅	□	□	□	⊞	□	□	□	□	□	□	□	□	■	■	□	⊞	■	■	■	■	□	■	■

pecially when the database is a graph of interconnected entities with a structure that is not as homogeneous and regular as in traditional business applications." [AGO95, p.4] Fibonacci is a general-purpose programming language that also provides transaction mechanisms. Side effects on persistent and transient values are undone when transactions are reverted.

Architecture Fibonacci provides roles and behavior evolution as first-class concepts. The type system of Fibonacci is presented informally. Fibonacci distinguishes between *role types* and *object types*. Object types are mere placeholders; empty supertypes without properties and behavior. The only operations that are computed on object values (instances of object types) are equality operation, role casting, role inspection, and role extension.

In object-oriented languages, there normally exists *one* most-specialized type only, i.e., the most concrete sub-type the object is instantiated with. In Fibonacci, a single object may have different most specialized roles. For example, a `Person` role is extended by `Student` and `Employee`; the object value plays both roles at the same time.

The implementation uses a compiler that type-checks a Fibonacci program and generates the corresponding Persistent Hierarchical Abstract Machine (PHAM) code. PHAM is a stack machine using zero-address operations to manipulate data and closures such as compiled functions. In Fibonacci all information, not only data, but the complete runtime state including the types, are persistent. The runtime lowers the complex and specialized Fibonacci data structures into the primitive and general-purpose data structure supported by the persistent Napier store [Bro89]. An interpreter executes the compiled Fibonacci code and stores it as first-class data.

Programming Model Implementation In Fibonacci dispatch always depends on the object that receives the message. This means that different sub-objects may dispatch the same message to different methods. Fibonacci supports two lookup mechanisms: inheritance and message passing (super call). By inheritance, the super-role is searched for a method when the queried role does not define such a method. Message passing means that the sub-role calls the method of the super-role directly via super. Internally the objects represent the roles they play in a Directed Acyclic Graph (DAG), where the object value is the root and roles are the leaves. Objects must be accessed via their roles only. If a message is sent to a most specialized role of an object then upward and double lookup coincide.

Both upward and double lookup are two forms of *late binding* that we depicted in Figure 5.2. *Upward Lookup* means that the method is looked up first in the receiving

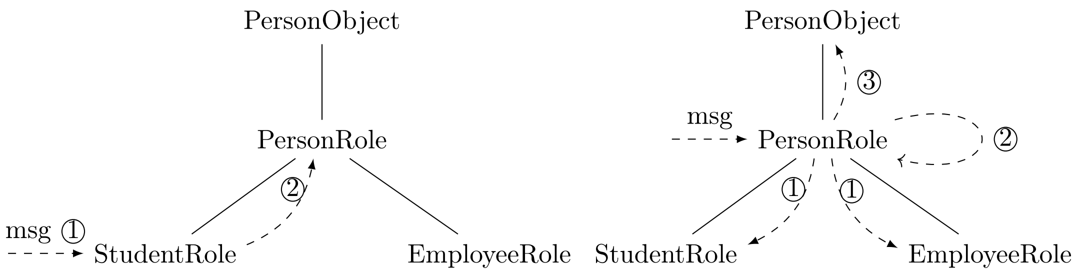


Figure 5.2: Upward lookup (left) and Double lookup (right) depicted on a role inheritance hierarchy according to Fibonacci [AGO95]. Dashed arrows with numbers represent the order applied using the lookup strategies.

role and then in its ancestor roles. The syntax in Fibonacci for upward lookup is `object!message`. On the other side, there is *Double Lookup* where the method is looked up first for all descendants of the receiving role, then in the receiving role, and, finally, in its ancestor roles. The syntax in Fibonacci for double lookup is `object.message`.

Weaving or Mapping Implementation The ROP implementation directly captures role entities in the programming language. The literature did not specify how terms of the Fibonacci programming language are represented in the PHAM. It is also not specified how the PHAM represents the runtime state in the persistent store.

Context and Role Instance Management The language does not support context-oriented programming according to the definition of *Compartments* [Küh+14]. Role instances are directly managed in the internal DAG representation.

5.2.3.2 Chimera

Chimera [CM95] is a strongly typed object-oriented database language. It consists of an object-oriented data model and a database language that supports a declarative query language, data definition, and data manipulation primitives and operations. For this review, we concentrate on the solutions that arise from problems in multiple direct memberships of objects [BG95; GBB98].

Language Presentation Chimera also supports objects that belong to *multiple* most specific classes [BG95]. Though Chimera does not have reified contexts (compartments) it states to provide context-dependent dispatch. To achieve this it provides a *preferred* *class* approach combining static and dynamic information. The *argument specificity* approach only uses dynamic information closely resembling multiple-dispatch by taking into account the run-time types of the actual parameters of the invocation. The authors reference roles as the de-facto alternative to their approach [BG95, p.105]. While Chimera is not a role language itself the authors reference data models supporting roles as alternatives to realize context-dependent behavior.

Architecture In Chimera's [CM95] expressions, the type of each variable used in the expression is stated with class formulas using unary predicate symbols. Thus, the formula of a variable X of type T is $T(X)$ and declares that T is the *static type* (or *context*) of variable X. The scope of a type assignment for a variable is that of an expression. At execution time X is instantiated with a value of type T. If T is a class name the variable is instantiated with an instance of the class identified by T. At execution time that variable may also be instantiated with a subclass T' of class T. T' is said to be the *dynamic* type of variable X. The set of dynamic types of an object is constrained such that a variable's dynamic types must have a common ancestor in the inheritance hierarchy.

Programming Model Implementation While most object-oriented languages restrict objects to belong to a single most specific class, i.e., a single subclass, Chimera [CM95] supports objects that belong to *multiple* most specific classes [BG95]. This requires a more complex protocol for method lookup and dispatch since each class in the hierarchy may define a different implementation for the same method. The *preferred class* approach supports a context-dependent behavior by combining static and dynamic information to resolve method invocations. Dispatching is dependent on the *context* (or static type) and the preferred class of a variable that is referencing the object. To assign a preferred class to each object in each expression for method dispatching the total order of classes of the class hierarchy is used. This order is determined by the definition order of classes. The dynamic information consists of the set of most specific classes of the object which is determined at run-time.

Weaving or Mapping Implementation We could not find any information on that matter.

Context and Role Instance Management We could not find any information on that matter.

5.2.3.3 Object With Roles

Objects with roles[8] [GSR96] proposes the extension of class-based object-oriented systems to handle evolving objects.

Language Presentation The approach is embedded into the class-based object-oriented programming language Smalltalk [GR89]. Class hierarchies are complemented by role hierarchies in which a node represents a role type and the root represents the object. At the schema level, a role hierarchy looks like a class hierarchy—role types define a set of instance variables and methods. The difference to ordinary class hierarchies is that a subtype in a role hierarchy does not inherit the definitions of variables and methods from the supertype. Instances of a subtype and of the supertype that represent the same real-world entity are related by a `roleOf` relationship. In contrast to ordinary class-based inheritance, this permits an *instance* of a subtype to inherit properties and methods from the *instance* of the corresponding supertype. The approach is embedded into the Smalltalk dispatch mechanism where every message not understood by the former is delegated to the latter. [9] Thus, a real-world entity is represented by multiple instances each representing the same logical object in a different role which can answer messages collaboratively. By allowing different simultaneous views of an entity context-dependent behavior can be realized.

Architecture In Objects with Roles [GSR96], the role hierarchy relates role types in a tree-shaped hierarchy by a `roleOf` relationship. The root of the tree defines the invariant properties of the object. Other nodes in the tree define the properties an object may acquire and lose over time. To extend the Smalltalk programming language with roles the approach defines two classes—`ObjectWithRoles` and `RoleType`—that serve the purpose of meta-classes. Nodes in a role hierarchy are subclasses of the meta-classes. Relations between role types are not mapped into class inheritance but are captured in a property that is dynamically evaluated during dispatch. The approach supports the assumption of multiple instances of the same role type called *qualified roles*. Same-typed role instances are identified by the instance provided as a qualifier. For example, a `ProjectManager` is qualified by the `Project` supervised.

[8]The authors did not assign a name to the approach. We deliberately assigned a name that appropriately describes the approach.

[9]Smalltalk being a dynamically typed, class-based programming language, a class can define an exception handler that reacts to `messageNotUnderstood`, i.e., a call to a function that is not implemented in the class and can delegate the call to another instance to handle the message

Programming Model Implementation The approach unifies the ideas of class-based inheritance and object-based inheritance (also known as prototype-based inheritance). Class hierarchies are complemented by role hierarchies in which a node represents a role type and the root represents the object. The meta-classes `ObjectWithRoles` and `RoleType` form a MOP by defining methods to handle the membership of instances of subclasses of meta-classes. This is similar to how the *Role Object Pattern* (cf. Section 3.2.1 and Section 6.3.1) is realized in statically typed, class-based object-oriented programming languages. Entities are assigned a unique system-defined identifier and the entity identity is provided by the object at the root of the role hierarchy. Roles are identified by the unique system-defined object identity. That is, two role instances of the same entity are not equal given their different object identities. *Entity equivalence* is defined by two role-type instances having the same root objects, i.e., their entity identifiers being equal. Instances of a subtype and of the supertype that represent the same real-world entity are related by a `roleOf` relationship. This permits an *instance* of a subtype to inherit properties and methods from the *instance* of the corresponding supertype. Every message not understood by the former is delegated to the latter. The standard scheme for method dispatch is similar to *upwards lookup*. Besides handling `messageNotUnderstood` the MOP also provides means to cast an instance of a role type to another role type[10].

Weaving or Mapping Implementation The approach is directly embedded into the dynamically typed, object-oriented programming language Smalltalk [GR89]. The implementation uses a MOP. The exception handling methodology of Smalltalk based on `messageNotUnderStood` is used to realize the approach as an internal DSL.

Context and Role Instance Management Role instances are directly represented as objects in the Smalltalk runtime.

5.2.3.4 Split Objects

Split Objects [BD96; BD95] are proposed as an example of a disciplined and semantically founded use of delegation [Lie86].

Language Presentation The approach discusses *name sharing*, *property sharing*, and *value sharing* between objects. Name sharing means that two objects that share a

[10]For example, a message will fail if the role type or any instance of the supertypes do not implement a method, whereas the message `(studentSmith as: Employee) salary` will return the salary of Smith as an employee.

name will have a property under the same name. Property sharing implies name sharing. Thus, when two objects have the same property identified by the same name. Property sharing expresses viewpoints within objects. In Split Objects, delegation is used *inside* objects to express property sharing between different perspectives or viewpoints. Value sharing describes the sharing of property values. If both objects have properties identified under the same name their values will be equal. In the author's view inheritance is a sharing relation. Thus, inheritance in a class-based system is name sharing for variables and property sharing for methods.

Architecture Split Objects [BD96] overcomes the problems that arise in a pure object-based object-oriented system using delegation. Method dispatch is described with the delegation method in terms of name sharing, property sharing, and value sharing. An important property of split objects is that delegation can be used to achieve a per-viewpoint representation of a single identity of the real world.

Programming Model Implementation In delegation-based systems, declarations of property names and value declarations are done at the object level. Objects are linked together by delegation links. Given two objects o_1, o_2 linked together by a delegation link. When o_1 is asked for the value of a property named n which is not defined on the object o_1 itself then o_2 is asked for the value of that property. The relation to class-based inheritance is imminent. [11] The approach highlights how problems connected to property sharing stem from the fact that a pure object-based system requires dedicated support to *clone* an object. [12]

Messages sent to an object must specify the viewpoint by giving the identifier of the piece denoting it. A lookup is then performed, starting at the identified piece and eventually continuing to its ascendant pieces (upward lookup).

Weaving or Mapping Implementation The approach represents real-world entities as *split objects* which can be referenced and have an identity. Views of the real-world entity are represented as *pieces* which do not have their own identity but can be accessed through the encompassing split object by specifying the piece name.

[11] Given a subclass that does not override but inherits a definition the method m will not be found in the definition of the subclass but in the parent class.

[12] In later iterations of object-based object-oriented systems this is realized with *prototypes* that objects will be cloned from. In the prototype-based JavaScript [Moz24] programming language, one could, for example, change the value of a property of a prototype which will be visible to all its clones that do not provide a same-named property in the object itself.

Context and Role Instance Management The approach is embedded into the object-oriented SELF [US87] programming language. The language runtime handles role instances.

5.2.3.5 Role Components

Role Components [VN96c; VN96b; VN96a; VN95] is an approach that focuses on a closer mapping from responsibilities in the collaboration-based analysis model to entities in the implementation. The approach improves the adaptability and code reuse of the implementation using Role Components. In [VN95] the authors highlight the similarity of the approach to the GenVoca [Bat+94] design pattern.

Language Presentation The approach is based on language features of the C++ programming language, especially the meta-programming capabilities using class templates. The language describes roles as refinements or extensions to a class' interface. In C++ this can be achieved by the means of class templates and inheritance as a logical glue for composing roles into classes. In that way, a role type can be defined by parameterizing the collaborators of that role. By instantiating the parameterized role types, the approach effectively generates specialized classes to fulfill the role. Listing 5.9 shows an example instantiation of Role Components showcasing how to realize a father-husband relationship.

Architecture Roles are represented by parameterized types realized by C++ class templates. Thus, a role has no independent identity and requires the instantiation together with the base class. Different template instantiations can provide different combinations of roles without modifying the role definitions themselves.

Programming Model Implementation Different template instantiations can provide different combinations of roles without modifying the role definitions themselves. While this allows to handle various orders of inheritance it also increases the responsibility of the application designer. Because when composing multiple roles into a single class the order of the composition is important. When multiple roles contribute the same method newly composed roles override the methods of roles composed prior. The authors propose the roles/responsibilities matrix [Bur95] to aid system designers in the implementation process.

Weaving or Mapping Implementation The approach is embedded into C++ using the class templates language feature. Roles are mapped to parameterized types.

Listing 5.9 An example of Role Components adapted from [VN96c]. The father-husband relationship is implemented by initializing C++ class templates.

```
template<class ChildType, class MotherType, class SuperType>
class FatherRole: public SuperType {
  ChildType* child;
  MotherType* mother;
  ...
}
template <class WifeType, class SuperType>
class HusbandRole: public SuperType {
  WifeType* wife;
  ...
}
class Father1: public HusbandRole<MotherClass, emptyClass> {};
class Father: public FatherRole<ChildClass, MotherClass, Father1>{};
```

Context and Role Instance Management Role instances are instantiations of parameterized types. There is no automatic memory management. Freeing allocated memory is the responsibility of the application designer.

5.2.3.6 DOOR

Dynamic Object-Oriented database programming language with Roles (DOOR) [WCL96; WCL97a; WCL97b; WC98] is a dynamic object-oriented database programming language with role extension to address dynamic type changes of real-world entities over time. While migration is often represented by changing the object that represents the entity this creates problems such as dangling references and does not allow the presentation of historical information.

Language Presentation At any point in time, entities are represented by their instance of the class and the instances of the roles the object currently plays.

The approach is implemented in Scheme [SS98; SG93] a dynamically typed functional Lisp[Bob+88] dialect. It uses a Metaobject Protocol and meta-functions that directly allow it to represent role-oriented programming constructs. An interesting idea in DOOR is that the model makes a distinction between the *dropping* or (temporary) *suspension* of a role [WCL97a].

Architecture DOOR is implemented as a MOP with meta-functions that directly allow it to represent role-oriented programming constructs.

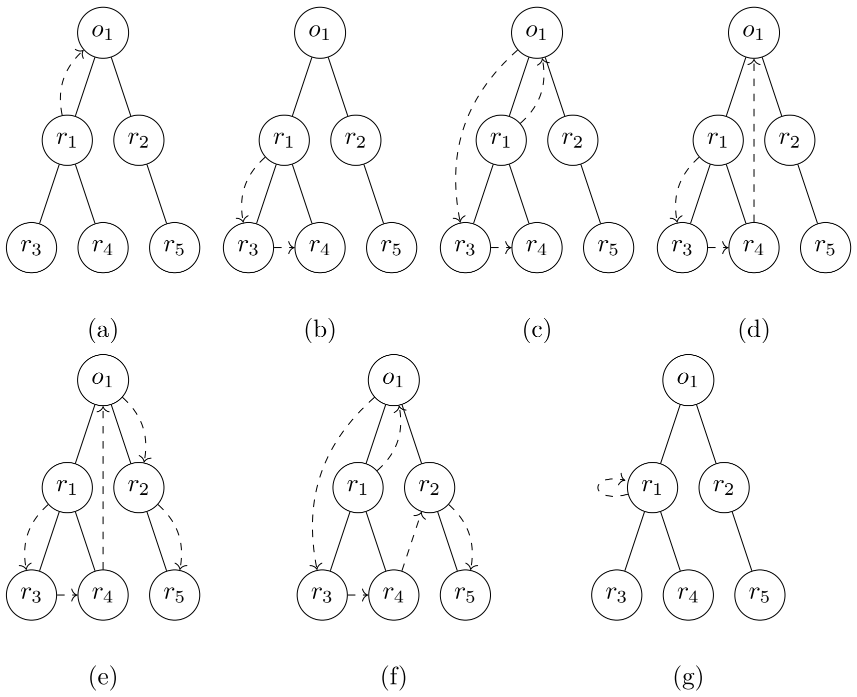


Figure 5.3: The different method lookup schemes for $o_1!r_1 \Leftarrow m$ provided by the MOP in DOOR adapted from [WCL97a, Fig.4]. The schemes differ in their encapsulation and data-sharing properties but provide higher flexibility. The delegation path is depicted with a dashed arrow: (a) $r_1; o_1$. (b) $r_1; r_3; r_4$. (c) $r_1; o_1; r_3; r_4$. (d) $r_1; r_3; r_4; o_1$. (e) $r_1; r_3; r_4; o_1; r_2; r_5$. (f) $r_1; o_1; r_3; r_4; r_2; r_5$. (g) r_1.

Programming Model Implementation In DOOR the role model is formed by introducing the role class hierarchy (*played-by* relationship) into the object class hierarchy (*is-a* relationship) where roles and classes are orthogonal to each other. the role class hierarchy is formed by the *played-by* relationship and the object class hierarchy is represented by the *is-a* relationship. The difference between roles and classes is found in their object identities. The *is-a* relation results in a class having the same object identifier as its superclass. This is not the case with the *played-by* relation between a class and its roles; roles have no globally unique identifier than their players do. Roles in contrast to ordinary objects are never referenced by their object identifier but by their role (class) name.

The implementation uses delegation to keep the program free of type errors, i.e., suppose the instance `e` of the role type employee played by a person `p`. A person has the property `sex` such that `sex` is an attribute of persons but not of employees. Then `sex(e)` would be a type error which is fixed by delegating the evaluation to `played-by(e)`. Figure 5.3 shows the implemented method lookup schemes implemented as MOP. The implementation provides support for seven pre-defined lookup schemes [WCL97a]. The schemes differ in their encapsulation and data-sharing properties but provide higher flexibility. The default scheme is described as an upward lookup in Figure 5.3 (a). The authors present a formalization of the type system based on ad-hoc polymorphism and *function overloading* [WC98].

Weaving or Mapping Implementation The DOOR data model is implemented in Scheme. Objects including their roles are represented as language objects, for example, a role is represented as a quadruple capturing the role class name the object is an instance of, the properties of the instance, roles played by the role itself, and methods defined for the role object. An object, on the other hand, is a quintuple which in addition defines an object identifier that identifies the instance inside the system. Method invocation is represented by a *path expression*. To express the invocation of an attribute `id#` of a particular role DOOR defines a role selection criteria using the pipe operator `|`. For example, to access the `ClubMember` role of the object `peter`, the DOOR statement is `peter!(ClubMember|clubname = "CS Club").id#`.

Context and Role Instance Management The Scheme runtime manages instances. We could not find any more information on that matter.

5.2.3.7 Darwin/Lava

Lava [CKC99] describes the implementation of the Darwin model [Kni99; Kni94], which is presented in the technical report aptly titled "*Objects don't migrate!*" [Kni96]. The conceptual model introduces semantics for roles and objects with roles.

Language Presentation The approach based on object-based inheritance is implemented as an extension of the class-based, object-oriented Java programming language [Kni94]. The language Lava [CKC99] extends the Java language specification by introducing new keywords, and modifiers, changing semantics in existing definitions. For example, the keyword `mandatory` can be applied to delegation links if its value must be non-null, or `<-` which denotes explicit delegation of a method such as `parent<-msg(arg)`.

Architecture Lava is compiled into Java Bytecode [Kni00, Chap. 7].

Programming Model Implementation Darwin models a conceptual object as the set of all its simultaneously existing roles. An entity of the real world can be described by a single *essential role* (similar to a *natural* [Küh+14]) and a set of *transient roles* which can be required and abandoned. A class hierarchy in Darwin is an orthogonal combination of inheritance and role-playing. On the type level, a relationship between the essential concept and its potential roles can be defined (cf. *fill* relationship [Küh+14]), and at the same time its static partitions are defined by inheritance.

Each part of the entity presents the full (conceptual) entity perceived from the perspective of the role. During the interaction of entities one of its roles typically dominates the other roles.

Roles can be organized in a partial order having the essential roles as the greatest element. Inheritance is defined w.r.t. that order; a role that is smaller (bigger) concerning the partial order is called a subrole (superrole). Incomparable roles are called sibling roles.

A subrole r is a specialization of a superrole r' and may override methods (behavior) of r'. Furthermore, r can use methods of sibling roles or its own subroles, but cannot modify their semantics. The former is achieved by delegating to the superrole while the latter is achieved by consulting the subroles and sibling roles. An informal discussion of the operational semantics of Darwin—especially on the implications of applying the static delegation features in a dynamic setting where delegation links can be reassigned at runtime—can be found in [Kni00].

Weaving or Mapping Implementation The implementation realizes delegation methods by using a new method signature, for example, `T msg(`T_1` arg)` is compiled into `T $msg$(...receiver, `T_1` arg)`. To be compatible with external code the compiler generates bridge methods. A bridging method replaces an existing method and calls the newly generated method which has a different signature. Accessing properties is also realized via method calls. For a property access the methods `arg.$getvar$()` and `arg.$setvar$(val)` are generated.

Listing 5.10 The BNF rule extension to define mixin layers in Java implemented in [BLS98]. Non-terminals are enclosed in arrow brackets, and optional elements are enclosed in brackets.

```
<layer_definition> ::=
  layer <layer_name> (<param_list>)
    realm <realm_name> [<super>] {<declaration_list>}
```

Sub-typing as described in the Darwin model is achieved in the generated code by adding `implements` declarations to the child class and eventually creating a suitable Java interface for the parent class. All methods of the inheritance hierarchy are added to the interface according to [Kni00, Table 7.5]. Finally, the name of the original declared parent class is replaced by the name of the generated interface in all type declarations.

Context and Role Instance Management The approach does not define objectified contexts. The implementation statically compiles roles into the inheritance hierarchy.

5.2.3.8 Mixin Layers

Mixin Layers [SB02] is an implementation technique for *collaboration-based* or role-based designs building on modular large-scale refinements. Fragments of classes and functions that–when composed–yield fully-formed classes; an area of research that is heavily applied in software product lines. The approach was applied in the Jakarta Tool Suite to implement DSLs [BLS98] and is heavily inspired by GenVoca [Bat+94].

Language Presentation The approach presents *mixin layers* as an extension to the Java programming language with language constructs to support collaboration-based design. The design is influenced by the OORAM approach [RWL96] where objects participate in collaborations via their roles. Roles fulfilled by an object inside a collaboration only affect the collaboration and will not propagate throughout the whole application. As shown in Listing 5.10 the extension of Java provides specific language constructs to define mixin layers [BLS98]. The keyword `layer` is analogous to `class` but defines a mixin layer. The keyword `realm` specifies interface conformance for mixin layers which can be understood analogous to Java's `implements` keyword.

Architecture A refinement is a functionality addition to the program that introduces a conceptually new service, capability, or feature. The approach considers large-scale refinements, where a single refinement alters multiple classes of an application. It is exhibiting *cross-cutting* such that the refined entities must be updated simultaneously and consistently.

The approach considers a static composition of collaborations, where roles played by an object are uniquely determined by its class. The implementation uses *mixins* which, in the general case, are *abstract subclasses* [GC90], representing a mechanism for specifying classes that eventually inherit from a superclass without specifying the concrete superclass at the mixin's definition. *Mixin layers* represent collaborations as outer mixins that encapsulate roles as inner mixins. A concrete instantiation of a mixin (layer) creates a concrete realization of the mixin. We see similarities shared by mixin layer and compartments (cf. [Küh+14]).

Programming Model Implementation Static mixins can be directly represented in C++ using template parameters for super-types. To ease the discussion, mixin layers are discussed in the context of C++ [SB02]. The extension of the Java language provides language constructs to define mixin layers but not general-purpose mixins. Each instantiation of a mixin layer is mapped into a specific Java class. Inner classes are used to represent inner mixins.

Weaving or Mapping Implementation The filling of a role inside a collaboration is statically mapped by instantiating a mixin or mixin layer.

Context and Role Instance Management Because mixins are statically instantiated there is no extra need for instance management.

5.2.3.9 Chameleon

Chameleon [GØ03; Gra03a; Gra03b; GØ02] is a role model that has similarities with AOP. The authors postulate that the role mechanisms are stronger than aspects because the former is a per-object mechanism while the latter is a per-class mechanism. The authors proposed the following terminology: base classes, also known as naturals [Küh+14], are named *intrinsics*. *Constituent methods* are role methods that override or introduce behavior in intrinsics.

Language Presentation The approach extends the Java programming language with a dynamic extension mechanism that is transformed into plain Java. A simple snipped

Listing 5.11 A code snipped from Chameleon adapted from [Gra03a]. `I` defines an intrinsic, `R` declares a role of `I` and defines the constituent method `bar`.

```
class I {
  void foo(int i) { bar(i);}
  void bar(int j) {...}
}
role R roleifies I {
  void bar(int k) {...}
}
```

of a Chameleon program is shown in Listing 5.11 where a role `R` defines a constituent method `bar` for intrinsic `I`.

Architecture The prototype has been implemented in OpenJava [Tat+00] which provides extensive support for program transformation. The approach resembles Composition Filters [ABV92] by filtering all access to intrinsics via role manager meta-objects. This is achieved by renaming the original methods of an intrinsic, replacing the original method with a same-named generated method that forwards to the role manager. To realize field access for constituent methods and roles access methods are generated.

Programming Model Implementation Constituent methods serve the same purpose as aspects in the AOP language AspectJ [Kic+01]. Roles extend players to get access to methods via inheritance mechanisms.

Roles can be used without the use of role-specific references. Constituent methods cannot be invoked explicitly but by their role manager who is in complete control of dispatching in the application. That way the aspect-oriented mechanisms of before, replace, and after can be delivered.

The delegation mechanism [Lie86] is implemented by *overloading* the methods of an intrinsic with extra arguments for *self*. The newly generated method for the intrinsic will call the overloaded method attaching `this` to the call. All implicit method calls will be transformed to call the passed *self* reference. Figure 5.4 depicts a simplified sequence diagram of method resolution in Chameleon.

Weaving or Mapping Implementation Access to intrinsics is filtered via role manager meta-objects. This is achieved by transforming all methods to consult their role

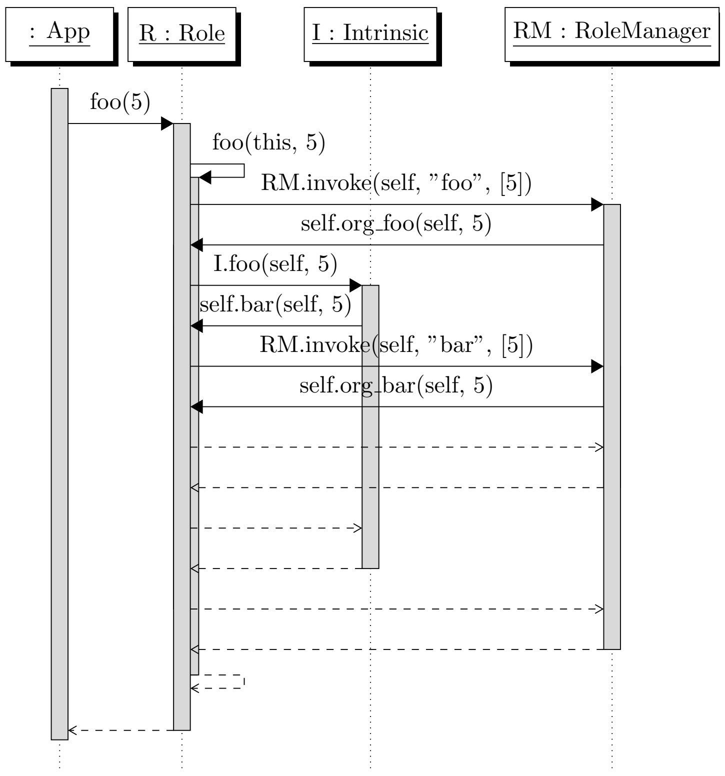


Figure 5.4: Simplified UML sequence diagram of method execution of the Chameleon program depicted in Listing 5.11. Adapted from [Gra03a].

manager.

Context and Role Instance Management The approach uses a meta-object role manager for bookkeeping role objects and role-playing. Each object has exactly one role manager attached to it.

5.2.3.10 Lasagne/J

The Lasagne model [Tru+01] and its implementation Lasagne/J [JT03] provide refinement operators for component-based systems as dynamic extensions to a core system. A subset of Lasagne/J has been formalized in [Moo+05]. The approach is highly influenced by the Role-Object Pattern [Bäu+97] and similar patterns such as the Decorator pattern [Gam+95]. Roles are represented by (dynamic) wrappers that attach new services to core component instances.

Language Presentation The Lasagne model [Tru+01] and its implementation coined Lasagne/J [JT03] provide system-wide and consistent refinements; cross-cutting concerns in the terms of AOP. A subset of Lasagne/J has been formalized in [Moo+05].

A collaboration is an n-ary interaction between objects. A dynamic adaption of collaborations is the change in the interface of participating objects dynamically widening the component's types. The subset of extensions that is selected depends on contextual properties [BDD00; De +05].

Architecture The model and implementation is realized with mixin-like wrappers [GC90]. Wrappers suffer from object identity problems [Höl93] and may result in object schizophrenia if not used correctly [CJ02; Her10]. Because wrapper-based designs can only support customization of a single component at a time the Lasagne model proposes a generic dispatch mechanism in the runtime component. This mechanism interprets an externally specified wrapper composition logic encapsulated in a *composition policy*. This is achieved by augmenting the object-oriented programming model with a *component identity* that unifies the constituents of a component and its decorating wrappers. An *extension identifier* binds a unique name to an extension. Extensions define the wrapper definitions to their members. The composition logic specifies the extension identifiers that must be applied to specific sets of collaborators. The composition policy is attached to each message send and may be updated for each ingoing or outgoing message of each component rather than being scattered across the program code.

Programming Model Implementation Mixin-like wrappers support modular customization of existing components of an application where each wrapper add behavior and state to its core component instance. The composition logic encapsulated in a composition policy can be customized on a per collaboration basis. The run-time component model supports dynamic construction of the wrapper chain by introducing a generic dispatch mechanism called *variation point*. At each variation point of a component instance the policy is interpreted and message dispatch redirected accordingly [TJJ00].

Weaving or Mapping Implementation The implementation Lasagne/J [JT03] is based on Java. Components and wrapper are implemented as a Java class. The core component is bound to its role instances at run-time where the wrapper instance holds a reference to its core component instance. Each call to the role interface will forward to the inner core component. The implementation adopts the mechanism to request role instances based on the Role-Object Pattern by querying the component instance using role instance specification objects. The role instance specification objects are the extension identifiers represented by strings.

Context and Role Instance Management Instances are managed via the component and wrappers. Addition and removal of roles is managed via the execution of policies.

5.2.3.11 Warped Hierarchies

Domain modeling often results in taxonomies of entities that could be represented by class hierarchies. In Warped Hierarchies [VDD04] it is found that when doing *role modeling* the entities follow the standard hierarchical taxonomy but the corresponding code demands the reverse version of this hierarchy. These warped hierarchies cannot be implemented in class-based object-oriented languages.

Language Presentation Warped Hierarchies is an approach build in Self based on the principles of multiple inheritance, shared parent objects, and copy-down techniques. Managing the roles of an object is achieved by applying meta-programming techniques provided from the Self language.

Architecture The approach is embedded in Self using multiple language features of the prototype-based object-oriented programming language. The authors make use of delegation to form role hierarchies but distinguish between where *data* and *behavior* is provided by an entity. The approach makes use of Self `trait` objects to provide behavior and `prototype` objects to hold data and the use of careful parent links (delegation links), copy-down, and reverse copy-down operations.

Programming Model Implementation Data and behavior is implemented separately. For each role a prototype object is created that keeps the data. A trait object of that role keeps the behavior. The role prototype is assigned a `parent*` link to the trait object of the role which is assigned a `parent*` link to the trait object of the natural.

Weaving or Mapping Implementation The dynamic start of playing a role is achieved by dynamically adding a slot to the prototype of the role-playing object which is assigned a data parent link to the role prototype. Removing the role is achieved by deleting the slot from the role-playing prototype.

If multiple roles provide an implementation for the same method the approach uses Self's meta-programming capabilities and implement a proxy that sequentially invokes the method on all roles currently played by that object, i.e., all data parents available.

Context and Role Instance Management Instances, either roles or naturals, are created as copies from their respective prototypes.

5.2.3.12 Java Role Package

The Java Role Package [ST04] is concerned with how strongly-typed programming languages such as Java or C++ can effectively implement role hierarchies and thus support the representation of evolving objects. It succeeds *ObjectsWithRoles* [GSR96] that introduced roles to the object-oriented programming language Smalltalk.

Language Presentation The Java Role package introduces a set of Java classes to support handling of evolving objects without modifying the semantics of Java itself. To reflect semantically correct usage of roles at runtime it requires role casting to request the correct role of a role-playing object before role-specific behavior can be invoked.

Architecture The Java Role Package defines role types in a role hierarchy. The difference from a role hierarchy to class hierarchies using inheritance is that by role-playing the role does not inherit the definitions of instance variables and instance methods from the supertype. It acquires a *roleOf* relationship to the corresponding role-playing object. Typically a role type can only be played a single time. If a

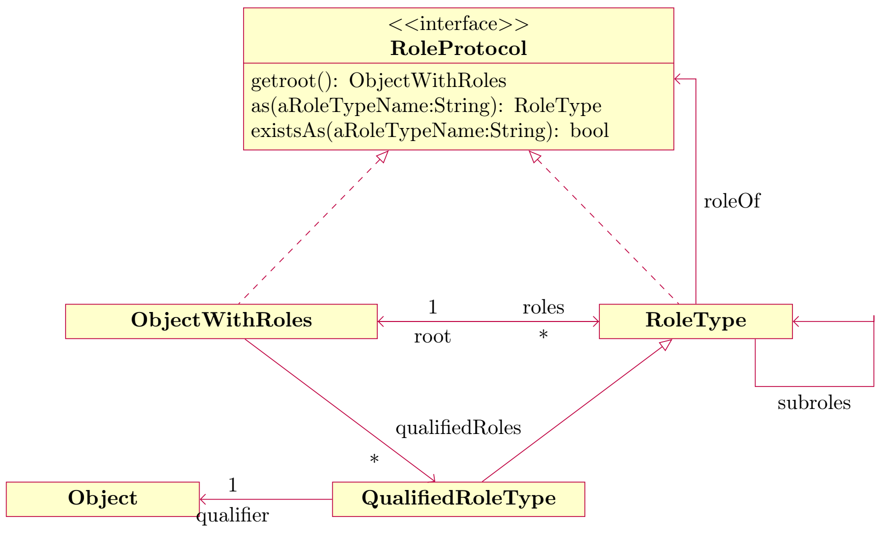


Figure 5.5: The Java Role Package meta-objects adapted from [ST04]. Naturals are supposed to extend `ObjectWithRoles` while roles must extend `RoleType`.

one-to-many relationship between the role player and its associated roles is required the *qualified role* must be used. A qualified role is a role type that uses a qualifier for precise identification. The MOP provided by Java Role Package is shown in Figure 5.5. Naturals are supposed to extend `ObjectWithRoles` while roles must extend `RoleType`.

Programming Model Implementation The approach uses a MOP approach by predefining a protocol and metaclasses. Domain classes must inherit from these metaclasses to inherit the protocol interface.

Weaving or Mapping Implementation The approach uses inheritance and a MOP to map role hierarchies into Java.

Context and Role Instance Management Instances must be managed by the package user. All maintenance operations must be programmed explicitly. The implementation uses hash tables to keep track of roles making its performance for role checking unaffected with the growing sizes of role hierarchies [SE06].

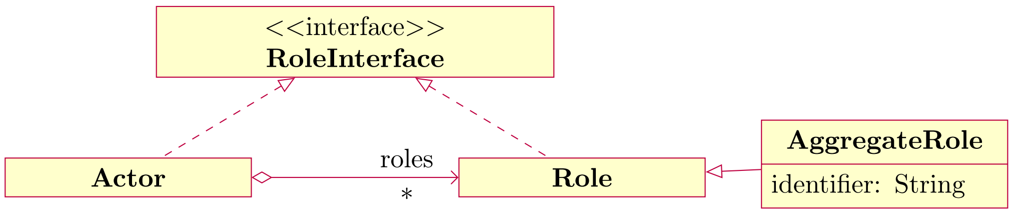


Figure 5.6: The JAWIRO meta-objects adapted from [SE06]. Naturals are supposed to extend `Actor` while roles must extend `Role`.

5.2.3.13 JAWIRO

Java With Roles (JAWIRO) [SE04; SE06] presents a library implementation to enrich the class-level inheritance from Java with instance-level inheritance using roles.

Language Presentation The approach extends the Java programming language to introduce object-level inheritance next to the existing class-based inheritance. Figure 5.6 shows the set of JAWIRO meta-classes that applications must implement or extend.

Architecture Role types and natural types have to implement a common interface [SE04]. The implementation uses a consultation mechanism where the implicit `this` parameter points to the object the message call has been forwarded to (cf. Figure 2.2). It provides a set of meta-classes that applications must implement or extend.

Programming Model Implementation The implementation defines an API based on the Role-Object pattern [Bäu+97]. Thus, role instances may be requested from a role playing object and must be casted to the requested role type. Method execution happens as usual by calling the method on the returned role objects. In the presence of multiple inheritance on object-level the implementation reverts to reflection in order to solve the typing ambiguity created by the resulting diamond inheritance.

Weaving or Mapping Implementation The role model is implemented using Java classes. Multiple inheritance on object-level is implemented using reflection.

Context and Role Instance Management Instances must be created, added, suspended, and removed by the programmer themselves.

5.2.3.14 Role Metaclass

The approach proposes a comprehensive implementation for the role relationship [DPZ02] using metaclass mechanisms [DPZ04]. The implementation depends on the metaclass approach provided by the VODAK modeling language (VML) [Kla+92].

Language Presentation The approach is based on the VODAK modeling language (VML) [Kla+92].

Architecture A metaclass `ObjectRoleClass` captures the generic semantics of natural types and role types engaging in role relationships. It is used to instantiate application classes that earn means for defining and querying the role relationship. It allows to define or delete instances according to the semantics of the role relationship. On the instance level it provides the methods for establishing or deleting links between objects.

Programming Model Implementation The implementation is based on metaprogramming capabilities of the VODAK modeling language. The defined metaclass `ObjectRoleClass` provides meta-functions for managing objects with roles. Natural types or role types must extend this metaclass. The VML does not support predicated execution. Predicates defined in the role relationship are implemented as boolean functions that return whether a predicate holds. The predicate functions will be invoked before or after a role is added or removed.

Weaving or Mapping Implementation Natural types and role types are instances of the generic metaclass `ObjectRoleClass`.

Context and Role Instance Management Instance management is provided by the metaclass `ObjectRoleClass`. The programmer has to invoke the appropriate meta-functions.

5.2.3.15 Rava

Rava [Che+06] is an extension to the Java programming language. It uses the Role-Object Pattern [Bäu+97] combined with the Mediator pattern [Gam+95].

Language Presentation The approach extends the Java programming language with new language constructs and provides a transpiler that translates the concepts of Rava into their equivalent Java representation. Language constructs added comprise keywords to declare roles and their players as well as keywords to annotate role invocations and access to properties of a role's player.

Architecture Core objects in Rava become Mediators in the Java representation. Role objects do not provide a custom constructor because they can not be instantiated independently but only from within the Mediator implementation.

Programming Model Implementation The approach uses a transpiler which translates Rava code to Java code. Therefore it has to visit the Rava program and translate concepts of Rava to equivalent Java representations. For roles, for example, a role interface is added and maintenance code is generated. References to a role's core are converted to the respective core instance. Besides the elements found in the Mediator pattern [Gam+95] the mediation works using the Role-Object Pattern [Bäu+97].

Weaving or Mapping Implementation Players and roles are represented by Java classes. The approach employs a single mediator per *player type*. Thus, the mediator instance has to store the player-role relationship (i.e., fills relation [Küh+15]) for each instance of the respective player type it mediates.

Context and Role Instance Management The mediator manages the lifetime of role objects by instantiating role instances when necessary. The mediator manages player and role instances centrally.

5.2.3.16 JavaStage

JavaStage [BA12] is an approach that makes static roles known from role modeling available in Java.

Language Presentation JavaStage is an extension to Java that uses roles as first-class constructs. Role declarations are similar to ordinary class declarations. Roles may declare fields and methods. Contrary to many other models, in JavaStage classes state what role they play and roles state the requirements they impose on their players.

Architecture The approach only considers static roles, i.e., roles of a class that are active all the time and that can not be attached or detached from an object. They argue that static roles have been used in role modeling, e.g. used in the OORam method [RWL96], and offer higher reuse than dynamic roles. This is grounded on the proposal, that a design with higher separation-of-concern has more self-contained components that may be easier to reuse.

A class consists of the class itself and the roles it is assigned to. The interface of that class is defined by the union of methods defined in the class and the methods defined by its roles. A role may not access members of a class and vice versa but may require the class to provide some interface. A role can state requirements the player must fulfill. By introducing generic type parameters and a `requires` clause that lists required methods a player type to be bound must provide.

Programming Model Implementation To increase decoupling the model dictates that classes and roles only rely on interfaces to communicate or compose. Methods declared in the player class have precedence over methods declared in roles. If multiple roles contribute a method to the class the first that introduces the method has precedence. The `javac` compiler has been extended to compile JavaStage syntax to Java bytecode.

Weaving or Mapping Implementation Roles are represented as inner classes of their *player classes*. Role methods are added to the members of a class. Methods declared in the player class have precedence over methods declared in roles. If multiple roles contribute a method to the class the first that introduces the method has precedence. Conflicts introduced by methods with the same signature raise a warning in the compiler and have to be resolved by the developers.

Context and Role Instance Management Roles as inner classes will be instantiated as soon as the instance of the role-playing class is instantiated. There is no extra instance management necessary.

5.2.4 Relational Role-Oriented Programming Languages

5.2.4.1 Rumer

Rumer [BGE07; Bal11] proposes a relational programming language that explicitly and implicitly models collaborations between sets of objects. It allows for modeling the collaborations of objects in a mathematically rigorous way using relational algebra enabling reasoning about the program.

Language Presentation The relational programming language builds from the following primitives: The relational model considers classes as types and as *sets*. Such a set contains the objects which are instances of the type defined by the class declaration. A *relationship* is a type and a relation. It contains the object tuples that are instances of the type defined by the relationship declaration. The *participants* of a relationship are the classes. These participants of a relationship declaration may carry a name to indicate the conceptual *role* the particular class plays in the relationship.

Architecture The programming language is based on relational algebra. The language supports value types, class names, relationship names, object instances, and queries on sets. Rumer represents these elements using tables and maps to formalize the declarations appearing in a program.

Programming Model Implementation The approach does not specify how the model is implemented.

Weaving or Mapping Implementation We could not find any information on that matter.

Context and Role Instance Management We could not find any information on that matter.

5.2.5 Contextual Role-Oriented Programming Languages

5.2.5.1 ObjectTeams/Java

The ObjectTeams (OT) programming model [Her03] and its most successful implementation ObjectTeams/Java (OTJ) [Her05; Her07] extends the Java programming language. The approach emerged from the aspect language Lua Aspectual Components (LAC) [HM01] of the Aspectual Components Model [HM00; MSL01] and has been tried in multiple programming languages [Vei14]. This thesis introduced OTJ in parts in Section 3.2.3. An in-depth account for OT and its implementation OTJ can be found in Chapter 7.

Language Presentation ObjectTeams is a representative of the class-based object-oriented approaches. The OTJ programming language extends Java with new syntactical features and metaclasses to define contexts and roles. The language and its informal semantics are presented in [HHM11]. The language bridges the statically typed, class-based object-oriented world and the dynamic, object-based world using elements from the AOP domain. OT introduces aspect-oriented and role-based concepts that are smoothly integrated with object-oriented concepts like inheritance and polymorphism. Highlighting the possible crosscutting nature of object collaboration the semantic of roles in ObjectTeams (OT) is similar to instance-local dynamic aspects.

The first-class entity context (cf. compartment [Küh+14]) in which roles can be played is represented using *teams*; a higher-order module for contained *roles*. The roles themselves are contained as inner classes within a team (see Listing 3.8 for an example).

Architecture In contrast to languages presented in the prototype-based approach, ObjectTeams is proposed as a type-safe programming language. The approach does not provide a formal definition of the type system but the language and its informal semantics are presented in [HHM11]. Types that are valid to fill a role can be declared with the `playedBy` relation restricting role-playing to the defined base type. The declaration of *bindings* is where the approach *gaps* the class-based and object-based world. A *callin* method binding intercepts the control flow at a method of the base entity and redirects it to a role method. That is how an object can *delegate* a method call to another object to handle it. The mechanism of bindings is based on aspect-oriented features that can come in different flavors such as before or after the specified method. When specifying a *replace* callin binding, the effect is the same as overriding a method in the context of inheritance. At runtime, there exist several means to define whether a binding is effective, i.e., whether or not the interception takes place. For example, a team *instance* can be activated or deactivated, which has the effect that all callin bindings of all contained roles are enabled or disabled.

Programming Model Implementation The implementation of OTJ uses ahead-of-time *compilation*. The approach substitutes the Eclipse Java compiler and extends the Java programming language backward compatible. The OTJ compiler provides role-specific static analyses and compiles to Java bytecode.

A team is mapped to a Java class which extends the OTJ meta-class `Team`. Roles themselves are contained as inner classes within these team classes. The framework uses envelope-based weaving [Boc+05] to realize the aspectual properties defined by callin definitions. The approach uses a MOP to realize the role semantics embedded within Java (see UML class diagram shown in Figure 7.1).

Weaving or Mapping Implementation The approach maps teams and roles to Java classes. A team is mapped to a Java class which extends the OTJ meta-class `Team`. Roles themselves are contained as inner classes within these team classes. The runtime provides Application Programming Interface (API) to realize the discovery of role methods. The code to realize the API calls is created by the ahead-of-time compiler and the weaver.

Context and Role Instance Management Caching of roles as an integrated part of role lifting mechanism [HM00]. The same mechanism later was implemented by Caesar coined "*wrapper recycling*" [MO02; MO03].

5.2.5.2 RICA-J

The Role/Interaction/Communicative Action ($\mathcal{RICA}$) theory, integrated relevant aspects of Agent Communication Languages (ACL) and Organisational Models to provide a metamodel centered around roles, interactions, and communicative actions. The metamodel is made available in the programming language RICA-J [SO04] to support the development of agent-based applications built on top of the Jade [BPR01] platform following the specification of Foundation for Intelligent Physical Agents (FIPA). Agent system specifications may be built using the concepts of communicative roles and interactions.

Language Presentation The $\mathcal{RICA}$ model captures *agents* whose behavior is manifested due to their assigned *roles.* The type of an agent is inferred by the collection of the agent's roles. The set of *actions* defines the tasks each role can fulfill. Roles that delegate their functionality are termed *enclosing roles.* Actions require the definition of input and output types for their parameters. The $\mathcal{RICA}$ theory, based on social-level analysis, distinguishes between generic and *social roles* representing the functionality of agents in *social interactions.*

While the interactions are responsible for letting agents assume their respective roles, the concrete behavior of entities and their roles is defined (and constrained) by *protocols.* Thus, a protocol regulates the behavior of a role in the context of an interaction. We see protocols as a representation of compartments [Küh+14].

Architecture RICA-J extends the Jade [BPR01] platform, a FIPA-compliant platform implemented in Java. The approach uses a Metaobject Protocol which is represented by Java classes. RICA-J entities must inherit the MOP to integrate into the framework. RICA-J takes a protocol-focused approach where interactions are started and may be aborted. The implementation checks whenever an agent is created or an event occurs whether a role must be abandoned or woken up.

Programming Model Implementation The implementation is based on an agent-based MOP. Metaobjects must be extended to gain access to agent and role functionality. Protocols define the behavior that is available on roles. Roles implement the specifications and are played by agents.

Weaving or Mapping Implementation Concrete implementations of agents, roles, and interactions are subclasses of RICA-J metaobjects.

Context and Role Instance Management Roles are bound whenever an interaction is started executing a predefined initialization method.

5.2.5.3 PowerJava

PowerJava [BBvdT06a; BBvdT06b] is a Java-based approach that defines roles to only exist due to the instance that plays the role as well as the context it is defined within, so-called *institutions*.

Language Presentation The approach presents *powerJava* and provides a transpiler that translates concepts of powerJava to Java. Roles are defined within an institution (cf. compartment [Küh+14]).

Institutions, players, and roles are directly defined using custom statements. The Java language is extended by *role casting* expressions that allow to cast role-playing objects to their roles.

Architecture We show a UML class diagram of the metamodel capturing the relation between roles and institutions present in powerJava in Figure 5.7. Institutions, roles, and players are translated into ordinary Java classes. powerJava uses delegation to access role methods but does not reveal how that translation is implemented.

Programming Model Implementation The approach separates the definition of roles and their requirements stated with a `playedBy <Interface>`. Interestingly, they reference interface types instead of concrete base classes in the `playedBy` clause which have to be implemented by their players. In the same manner as the definition of an

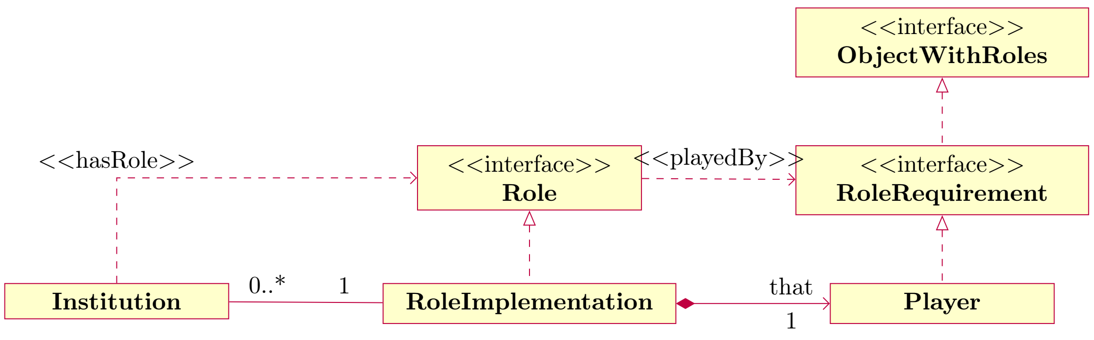


Figure 5.7: A UML class model depicting the relations of roles and institutions implemented in powerJava. Figure adopted from [BBvdT06b, Fig. 1].

interface, the definition of a role holds a set of function definitions. The implementation of the other side *realizes* those function definitions implemented as an inner class of an institution. Players of a role, on the other hand, have to *implement* the interface referenced in the `playedBy` clause.

Role-playing does not implicitly override the properties and methods of the player. To invoke methods from a role the player must be explicitly cast into the respective role. This can be achieved using a role cast expression that casts a player to the specified role.

Weaving or Mapping Implementation To keep the ontological foundation it requires the instance playing the role and the instance of the institution the role is defined within to instantiate roles. Role casting in powerJava conceals a delegation mechanism that is implemented in Java.

Context and Role Instance Management We could not find any information on that matter.

5.2.5.4 EpsilonJ, NextEJ, and Featherweight EpsilonJ

EpsilonJ [MT08; TUI05] is a role-based programming language realizing the Epsilon model [TUI07]. NextEJ [KT09] extends the Epsilon model and introduces new scoping rules. A formal model that captures the scoped role bindings based on the minimal core calculus Featherweight Java [IPW01] is presented in Featherweight EpsilonJ (FEJ) [KT10].

Language Presentation The programming language EpsilonJ extends the syntax of Java with role-specific syntactical elements. Collaborations are represented by a *context* which features several roles. Objects participating in the collaboration are called *player* and are represented by ordinary classes. Role binding is represented syntactically with a `bind` operator provided from the contextual roles.

Architecture A program written in EpsilonJ syntax is transformed into an equivalent Java program. While NextEJ [KT09] was never realized in a real implementation its semantics was captured in a minimal core calculus called FEJ [KT10]. The calculus is based on Featherweight Java [IPW01] and extends the calculus with context activation scopes known from the COP domain.

Programming Model Implementation The first implementation of EpsilonJ [TUI05] used the annotation features of Java. This approach is reported to "*resulting in the significant runtime overhead*" [MT08]. It was replaced by a transpiler which translated EpsilonJ directly to Java [MT08].

Weaving or Mapping Implementation Contexts, roles, and players are represented using classes. In EpsilonJ [MT08] static roles are represented by fields where non-static roles are kept in a vector; a dynamic list data structure provided by the Java standard collections. The dynamic role binding mechanism is implemented using role casting and delegation mechanisms. Thus, the role cast `(todai.Employer)sasaki).pay()` that casts the player `sasaki` to its role `employer` played in the context `todai` is translated to a search of the correct role instance filled by the player instance inside the given context instance. While a player could potentially play the same role multiple times (cf. Feature 4 in Table 5.1) it is not clear which role will be returned by a role cast.

Context and Role Instance Management Access to roles is implemented using collection operators such as iterators.

5.2.5.5 Scala Roles

Scala Roles [Pra08; PO08] is an implementation of contextual roles in Scala. It uses Scala's and Java's language constructs to hide split objects avoiding the problems of *object schizophrenia* [Har08].

Language Presentation The Scala Roles language is embedded into Scala. They claim to avoid the problems of *object schizophrenia* [Har08] using dynamic *compound* objects.

Architecture The approach is embedded in Scala. The approach uses *compound* objects to represent role-playing objects and their roles providing a single interface to the outside. A compound object is a product type consisting of the type of the player instance and the types of the role instances that are bound to it. A shortcoming is that the order of the component types determines the resulting type [Leu17b].

Programming Model Implementation Compound objects are implemented using Java's `Proxy` class which allows the creation of product types dynamically at runtime. The compound object consists of the player instance as well as arbitrary many role instances. Method lookup is implemented via delegation using reflection. A proxy delegates to either a role if that implements a method or the player instance otherwise.

Weaving or Mapping Implementation Contexts, roles, and players are represented using ordinary Scala traits.

Context and Role Instance Management There is no special management implemented. Role objects are collected by the garbage collector whenever the proxy object leaves scope.

5.2.5.6 SCROLL

Scala ROles Language (SCROLL) [LA15; Leu17b] enables on the one hand view-based programming with contextual roles as well as evolving objects embedded into Scala. A discussion on SCROLL in the context of roles in prototype-based programming languages can be found in Section 3.2.2.

Language Presentation SCROLL seamlessly embeds into Scala based on compiler rewrites, implicit conversion of arguments, and the ease of embedding domain-specific elements in the host language Scala. An example program written in SCROLL can be found in Listing 3.6. SCROLL provides a MOP where compartments must extend the `Compartment` class and roles are defined as inner classes of compartments. Objects that have to be treated as role-playing objects must be accessed with a preceding plus operator, e.g., `+joe.getNumber`, due to the implicit conversion applied.

Architecture SCROLL does not provide a type system. Arbitrary objects can become role-playing objects by being wrapped into a `Player` (meta) class, applying Scala's implicit conversions. Methods invoked on a player instance that are not defined on `Player` will be rewritten into a dynamic runtime invocation which triggers a search on the internal role-play DAG (cf. Listing 3.7). The approach requires that the searched method's signature must be implemented by the players and their roles. The dynamic lookup considers a later acquired role to precede any other role (cf. acquisition dispatch [Gra06]). A program may provide a user-defined method resolution algorithm.

Programming Model Implementation The implementation uses a combination of compiler rewrites, implicit conversion of arguments, and the ease of embedding domain-specific elements in the host language Scala. SCROLL provides a MOP where compartments must extend `Compartment` and roles are defined as inner classes of compartments. For example, in Listing 3.7 we show how compiler rewrites by the Scala compiler enable an opaque embedding of role dispatch in SCROLL.

Weaving or Mapping Implementation Compartments, roles, and players are ordinary objects. The roles played by an object are stored as a DAG inside the compartment.

Context and Role Instance Management The approach does not provide any special instance management.

5.2.5.7 LyRT

LyRT [Tai+16a; Tai+16b] is a Java-based library that proposes contextual roles to support unanticipated adaption of instances at runtime. Its purpose is to be used in a transaction-based environment where role acquisition is transactional [Tai+17].

Language Presentation LyRT provides a library to enhance Java with object-based dynamic adaptation mechanisms. To be a core object, for example, a user-defined class must implement the `IPlayer` interface which already provides implementations for role-specific functionality. The same applies to user-defined role classes. On the other hand, a user-defined compartment must extend the framework's `Compartment` class. The approach uses a registry that registers and manages the object adaptations. Therefore, each *class* must be reflectively registered. For example, to register a new player object with the registry we can use `Object ely = registry.newPlayer(Person.class)`.

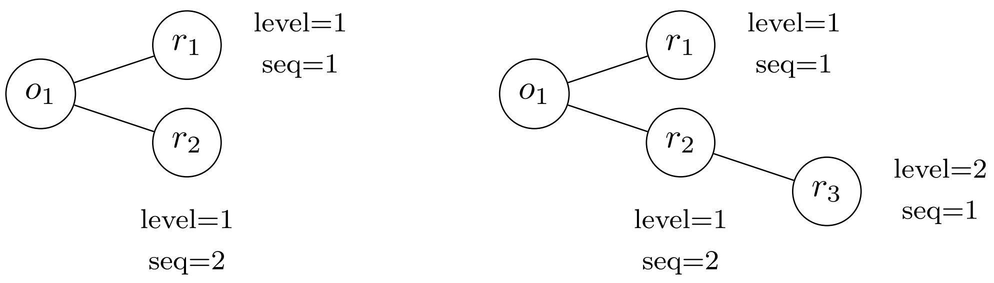


Figure 5.8: The tree spanned from the role-playing relation in LyRT. Method invocation uses levels and sequences to determine the target of a method invocation. Adapted from [Tai+16a, Fig. 3].

The role invocation uses a similar reflective mechanism which has to be actively used by the application developer. For example, the Java code using the registry to invoke a role method `registry.invokeRole(compartment, ely, "work")` realizes the invocation of `ely.work()`.

Architecture The approach does not provide a type system. However, in LyRT the role-playing relationship spans a tree where the player is the root and the roles are leaves. In the tree, each node is labeled. One label captures the *level* of a node, where *sequence* assigns a number to each sibling in order of acquisition. In Figure 5.8 the tree and its labels before and after the acquisition of role `r3` to role `r2` are shown.

Programming Model Implementation The approach demands that each compartment, role, and player has to register at the registry. The registry, like a relational database, keeps all relations between compartments, roles, and their players.

Weaving or Mapping Implementation The approach uses a library to manage classes and the linking of each class' provided methods at runtime based on a preview implementation of `invokedynamic` in Java 7. That is, all methods declared by a class are extracted and stored in a map to implement LyRT's dispatch mechanism.

Context and Role Instance Management The role-playing relation (cf. [Tai17, 4.1.2 Relations]) inside a compartment instance between a player and a role instance (or between role instances for deep roles) is stored in linked lists.

5.3 Conclusion

We reviewed a total of 25 role-oriented programming languages where we assigned 15 approaches to belong to the behavioral category, 1 approach to belong to the relational category, and 9 approaches to belong to the category of contextual role-oriented languages. While previous reviews concentrated on the features of role-oriented programming languages we compiled the implementation strategies starting from the concepts that surfaced in the role-oriented programming languages, ranging over the mapping strategies used to realize the concepts in a target language, and finishing with specifics in the management of the conceptual instances.

We agree with Kühn et al. that the "*evaluation indicates that more than half of the approaches were unaware of the possible features of roles or other related approaches*" [Küh+14]. Even more, we conclude that the features that are available in a target language influence the design space that approaches used to realize their features. For example, roles have always been considered a context-dependent view. However, since the inception of reified contexts (compartments) every approach used delegation-based dispatch. Furthermore, in each approach providing contextual roles (with LyRT being the exception) roles have been realized as inner classes of compartments.

6 Quantitative Aspects of Role-oriented Programming Languages

"Premature optimization is the root of all evil."
— Donald E. Knuth, 1974

In the previous chapter, we have seen that there is no single understanding of the role-oriented paradigm and the features that can be ascribed to implementations of this paradigm [Ste00a; Küh+14]. The review also unearthed the different design choices taken to realize those features. However, we found no in-depth assessment of the quantitative aspects of the different implementation techniques. The authors often qualified resulting implementations as "*slow*" or to "*suffer from a run-time performance overhead*" [Tru+01] missing a rigid evaluation of quantitative aspects. We mean that quantitative aspects of computation refer to the use of physical quantities, e.g., time and memory usage [Ald20]. This chapter will propose a quantitative analysis of contemporary role-oriented programming languages. The goal is to analyze the core properties of state-of-the-art role-oriented programming languages. Based on the quantities we evaluate these systems from different perspectives such as architecture, language design, and semantics. To this end, we create a synthetic benchmark that makes extensive use of features ascribed to the role concept. Based on this benchmark, we perform a cross-language comparison and discuss the results with a focus on performance, scalability, and memory management provided by the role-oriented runtimes.

6.1 Benchmarking Separation of Concerns Approaches

This section introduces benchmarks conducted in related approaches as well as other role-based programming languages.

A framework that enhances Java with roles is proposed with JAWIRO [SE04; SE06]. The framework has been compared to other design patterns and has been measured using micro-benchmarks. The benchmarks measure how fast operations on roles behave in case of an increasing number of role instances in the stored hierarchy.

EpsilonJ is a role-based programming language that enhances Java with roles [MT08].

This approach uses a transpiler to translate EpsilonJ programs to standard Java. The authors measured compilation and execution time to compare the effectiveness of the generated code against hand-written Java code. They conclude that compilation time is not a significant factor, but execution time was two times slower compared to the hand-written Java code.

ContextJS is a context-oriented extension to JavaScript [KLH12]. The author discusses optimization techniques applied to the host language JavaScript to improve concepts found in COP. To discuss these techniques a micro-benchmark to measure the execution time of layer activations and dispatches to (partial) methods was conducted. The conclusion is that most time is spent on dispatching.

In JCOP, a Java implementation of a Context-oriented Programming language, the dispatch has been implemented using the new `invokedynamic` bytecode instruction [AHH10]. The benchmark compares the proof-of-concept implementation based on `invokedynamic` bytecode compared to the unmodified JCOP.

There have been different approaches to measuring the performance, e.g., the usage of micro-benchmarks or small applications. While employing micro-benchmarks could highlight performance problems of an implementation they require rigorous handling due to the underlying Just-In-Time (JIT) compilers that many host languages use. In the rest of this chapter, we propose a quantitative analysis of contemporary role-oriented programming languages that takes the variability of JIT compilers into account.

6.2 Approach

To assess a role-oriented programming language its artifact must be available and functional. We gathered the publicly available artifacts of the approaches reviewed in Chapter 5. Of these approaches only ObjectTeams/Java (OTJ) [Her07], LyRT [Tai+16a], and Scala ROles Language (SCROLL) [LA15] provide an implementation of their compiler which remained functional.From those compilers, ObjectTeams/Java is the most mature, documented, and stable. Most role-based programming languages have been implemented as a Java library, e.g., SCROLL and LyRT, or use Java as a host language, e.g., OTJ. Therefore, we restrict the analysis to state-of-the-art role-based programming languages executed on the same platform; the Java Virtual Machine.

The benchmark suite we used to define, execute, and measure the benchmarks has been inspired by the "*Are we fast yet?*" benchmark harness [MDM16]. A class diagram showcasing the harness is shown in Figure 6.1. The classes are loosely coupled where benchmarks can be run given the appropriate commands from the command line to

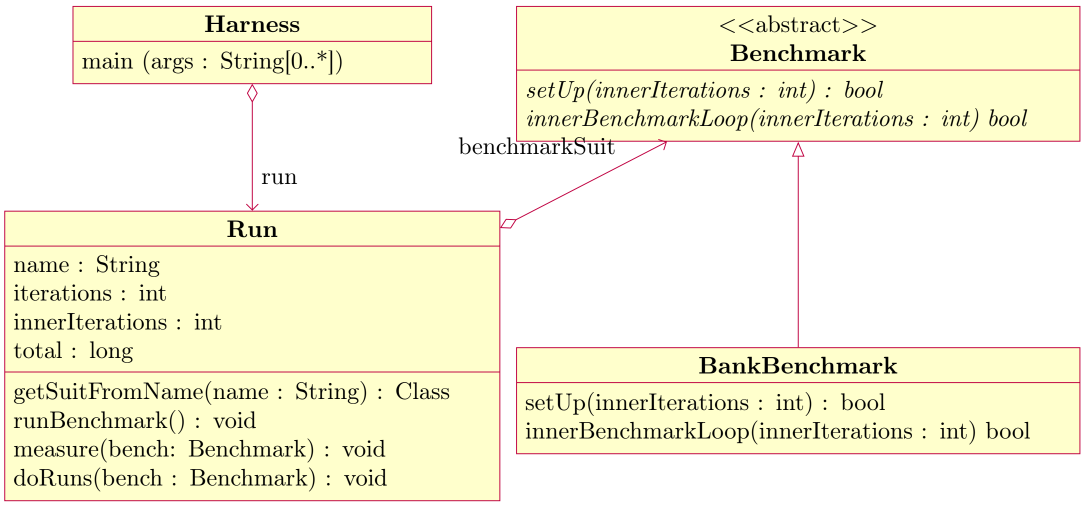

Harness
main (args : String[0..*])
run
<<abstract>>
Benchmark
setUp(innerIterations : int) : bool
innerBenchmarkLoop(innerIterations : int) bool
benchmarkSuit
Run
name : String
iterations : int
innerIterations : int
total : long
getSuitFromName(name : String) : Class
runBenchmark() : void
measure(bench: Benchmark) : void
doRuns(bench : Benchmark) : void
BankBenchmark
setUp(innerIterations : int) : bool
innerBenchmarkLoop(innerIterations : int) bool

Figure 6.1: UML Class Diagram of the Benchmark Suite used in this thesis.

the `Harness`. Specific benchmarks can be implemented by subclassing `Benchmark` and implementing the `setUp` and `innerBenchmarkLoop` functions. `Run` is supposed to set up the benchmark, execute, and measure the `innerBenchmarkLoop` execution time. To increase reproducibility we used `ReBench` [Mar18] to configure the benchmark executions and to parse the output of the benchmark suite. `ReBench` provides configuration files to start benchmarks with different implementations and stores the output in text files. We use `R` [R C18] to statistically analyze these files.

6.2.1 Benchmark Characterization

To benchmark the approaches we have chosen a model which has been used to qualitatively assess role-oriented programming implementations. Figure 6.2 shows a CROM-based role model of the simple use case of a scenario of a bank. There are `Person` classes that fill the role of a `Customer` in a `Bank`. A bank offers different types of `Accounts`, namely `SavingsAccount` and `CheckingsAccount`. The different types of accounts change the behavior of withdrawing money from an account or depositing money in an account. Besides, there is a `Transaction` compartment, where an account can either play the role of a `Source` or a `Target`, but not both at the same transaction defined by the role prohibition. For example, when an account plays the role of the `Source` in the transaction one is only allowed to withdraw money from that account. A more advanced version of the role model that uses role groups to model these restrictions can be found in [Küh+15, Fig.2].

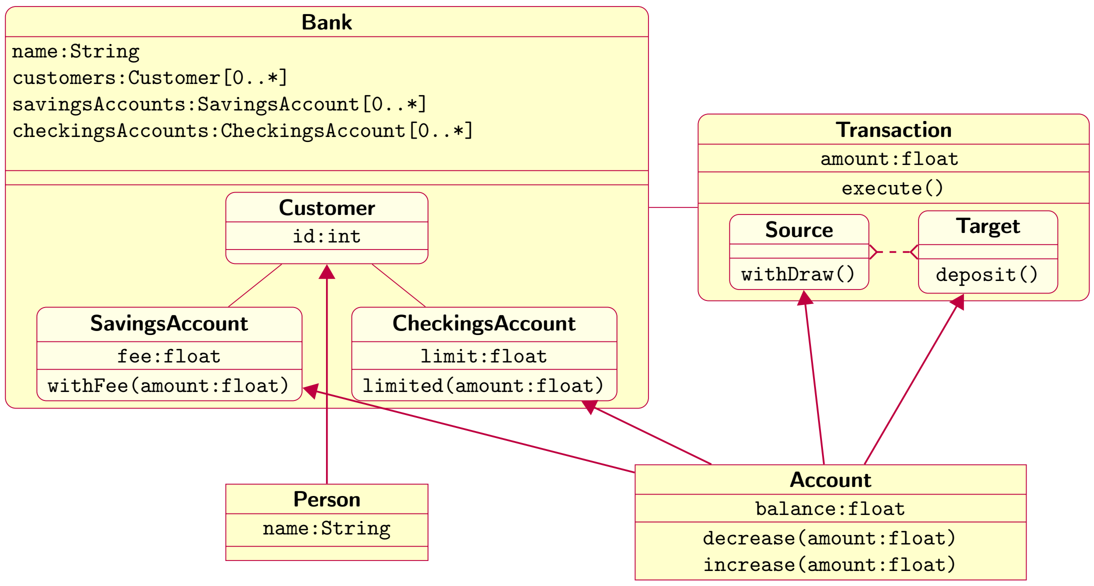


Figure 6.2: A CROM-based role model of a bank with different types of accounts. Customers can possess accounts that can play different roles across the compartments.

6.2.2 Methodology

Benchmarking the performance of a Java application is far from being trivial. The performance is affected by various factors such as the application itself, the input to the program, and the settings of the JVM such as the size of the heap, the selected garbage collector etc.[13] The performance of a single application can change from run to run because of the many sources of non-determinism such as the JIT compilation and optimization in the VM is driven by timer-based method sampling, garbage collection, thread scheduling and more. There exist two components to dealing with non-determinism in managed runtime systems. The first is the experimental design of the benchmark. The second is the statistically rigorous data analysis to deal with these

[13]The number of HotSpot JVM options in each version of OpenJDK varied between ~1500 - 1300 (including product, experimental, diagnostic, and developmental) [New21, Fig.4]

kinds of non-determinism [GBE07]. Existing tools such as the Java Microbenchmark Harness (JMH) [Ora23a] help with building, running, and analyzing benchmarks written in Java and other languages targeting the JVM. It allows the writing of fine-grained benchmarks at method, loop, or even statement level and supports preventing optimizations from distorting the result. However, the existence of such a tool does not prevent the benchmark designer from avoiding pitfalls when designing benchmarks [Dam+19].

Measuring *startup performance* has the goal of measuring how fast the JVM can execute a short-running program. These measurements typically include class loading and are affected by JIT compilation. The JVM will reach a *steady-state performance* once all relevant classes have been loaded and hot methods have been JIT compiled. In steady-state the sources of non-determinism are reduced and the execution suffers less from variability due to JIT compilation.

6.2.2.1 Data Analysis

Building a confidence interval requires a number of measurements $x_i, 1 \leq i \leq n$ from a population with mean μ and variance σ^2. The mean of these measurements $\overline{x}$ (sample mean) is computed as

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}. \tag{6.1}$$

The actual mean μ will be approximated by the sample mean $\overline{x}$. Additionally, we compute the confidence interval $[c_1, c_2]$ such that the probability of μ being between c_1 and c_2 equals the confidence level $(1 - \alpha)$. To compute the confidence interval of the mean we rely on the *central limit theory* which states that for large values of n (typically $n \geq 30$) [Lil05], $\overline{x}$ is approximately Gaussian distributed with mean μ and standard deviation $\sigma/\sqrt{n}$. Applying the central limit theorem we find that

$$\begin{aligned} c_1 &= \overline{x} - z_{1-\alpha/2}\frac{s}{\sqrt{n}}, \text{and} \\ c_2 &= \overline{x} + z_{1-\alpha/2}\frac{s}{\sqrt{n}}. \end{aligned} \tag{6.2}$$

The sample standard deviation s is computed as follows

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}}. \tag{6.3}$$

We use the statistical programming language `R` [R C18] to statistically analyze the resulting data.

Startup Performance Measuring startup performance has the goal of measuring how fast the JVM can execute a short-running program. These measurements typically include class loading and are affected by JIT compilation and interpreter performance [Mar+22]. To gather enough samples x_i we measure the execution time of p VM invocations, each invocation running a single benchmark iteration. Applying the preconditions to compute the confidence interval we find that $p \geq 30$.

Steady-State Performance The JVM will reach a *steady-state* once all relevant classes have been loaded and hot methods have been JIT compiled. In steady-state the sources of non-determinism are reduced and the execution suffers less from variability due to JIT compilation. To measure steady-state performance we follow a four-step methodology advocated in the literature [GBE07] which we shortly recite:

1. Consider p VM invocations, each running at most q benchmark iterations. Retain k measurements per invocation.
2. For each VM invocation i, determine iteration s_i where steady-state performance is reached. This means, that the coefficient of variation (CoV) of the k iterations ($s_i - k$ to k) falls below a preset threshold (e.g., 0.01 or 0.02).
3. For each VM invocation compute the mean $\overline{x_i}$ of the k benchmark iterations under steady-state:
$$\overline{x_i} = \sum_{j=s_i-k}^{s_i} x_{ij}. \tag{6.4}$$
4. Compute the confidence interval across the computed means $\overline{x_i}$. The overall mean is $\overline{x} = \sum_{i=1}^{p} \overline{x_i}$.

6.2.2.2 Experimental Design

The experiment is designed to measure startup performance as well as the steady-state performance of contemporary role-oriented programming languages. We are interested in the steady-state performance. To account for the dynamic behavior from the JIT compilation used by the JVM, the garbage collector, and the underlying operating system, the benchmark has been repeated several times. The experiments are designed to provide measurement points fulfilling the statistical requirements iterated previously. The input sizes and configuration parameters are listed in Table 6.1. We restart the experiment 3 times and in each execution, the problem is repeated 10 times. For the benchmark, we set the problem size in the range from 500,000 to 6,000,000 with a step size of 500,000, i.e., 12 different values. Thus, we get 30 measurements per problem size.

For each iteration, the harness will measure the execution time of the executed transactions. The benchmarks work on multiple different input sizes with varying numbers of `Accounts` as well as the number of `Transactions`. The different configurations are shown in Table 6.1. Thus, for problem size N, there are executed $N * N$ transactions, using N persons with $2N$ accounts (i.e., a `CheckingsAccount` and a `SavingsAccount`). The main loop of the benchmark (`innerBenchmarkLoop`) implemented in the language OTJ is shown in Listing 6.13.

To gather information about hot methods, memory consumption, and where the application spent its time every benchmark has been repeated once with activated profiling. Applications running for a longer period with a higher footprint on memory will result in more pressure on the garbage collector. This could make the garbage collector a dominant factor for the execution time. For a given role-based language, the memory footprint indicates how well the runtime manages resources.

The initialization of objects such as the instance of the bank, all the customers, and the accounts that are possessed by those customers are not measured. During each iteration, we executed `System.gc()` to release unused objects from the heap and to make the VM not collect garbage during the measured part itself. As such, the benchmark measures the combined execution time of creating a new transaction compartment, its activation, the binding of roles to this transaction, and at last the deactivation.

Table 6.1: Parameter configurations of the benchmarks. The upper part defines input parameters for the application, lower part defines parameters for the JVM.

Parameter	Values
runs	3
iterations	10
inner iterations	range of 500,000 to 6,000,000 with a step size of 500,000
VM heap size (–Xmx)	4G and 8G

Virtual Machines We run the benchmarks on different VM implementations to determine whether the role-oriented implementations behave differently in different environments. The Virtual Machines and their versions are listed in Table 6.2. The OpenJDK and GraalVM are distributed in binary form while we had to build OpenJ9 on our own. We used the suggested build parameters and did not tune any other setting from the build script. Unfortunately, the OpenJ9 build script already was built against the unreleased OpenJDK 21 while the others were built for Java 20. Due to restrictions of dependencies from Role4J, we were forced to use Java 1.8 to execute the benchmarks.

Table 6.2: The different Java Virtual Machines and versions used to execute the benchmarks.

Virtual Machine	Version	Platform
OpenJDK (HotSpot) 20.0.2	build 20.0.2+9-78	Linux x64
OpenJDK (HotSpot) 1.8.0	build 1.8.0_131-b11	Linux x64
Oracle GraalVM 20.3	build 20.0.2+9-jvmci-23.0-b14	Linux x64
Eclipse OpenJ9 0.40.4	openjdk 21-internal 2023-09-19	Linux x64

Hardware Platforms The experiments have been performed on an Intel® Core™i7-9700T CPU @ 2.00GHz with 32GB RAM. The operating system used was Ubuntu 22.04. We consider an otherwise idle and unloaded machine in our experiments.

To increase reproducibility we used `ReBench` [Mar18] version 1.2.0 to configure the benchmark executions and to parse the output of the benchmark suite. `ReBench` provides configuration files to define benchmark suits and experiments with different execution environments. It stores the output in text files which we statistically analyzed with `R`.

6.3 Role-based Benchmark Implementations

There is no single unified understanding of the role concept. Instead, existing role-based programming languages offer different features [Küh+14]. Next to different feature configurations, surveyed languages realize the same role concepts with different approaches. To account for language-specific restrictions of role features we provide two variants of the benchmark presented before either using *deep roles* or *flat roles* to represent the roles of accounts of a bank within a transaction. Each implementation provides the variants that are supported by the approach. For example, the Role Object

Pattern [Bäu+97] does not support contexts, but plain roles. In Role4J [Tai+16a], two compartments cannot be active at the same point in time. We tried, however, to keep implementations as similar as possible to enable a fair comparison. In the following, the different Role-oriented Programming implementations will be introduced. We present particularly interesting parts of the benchmark implementations.

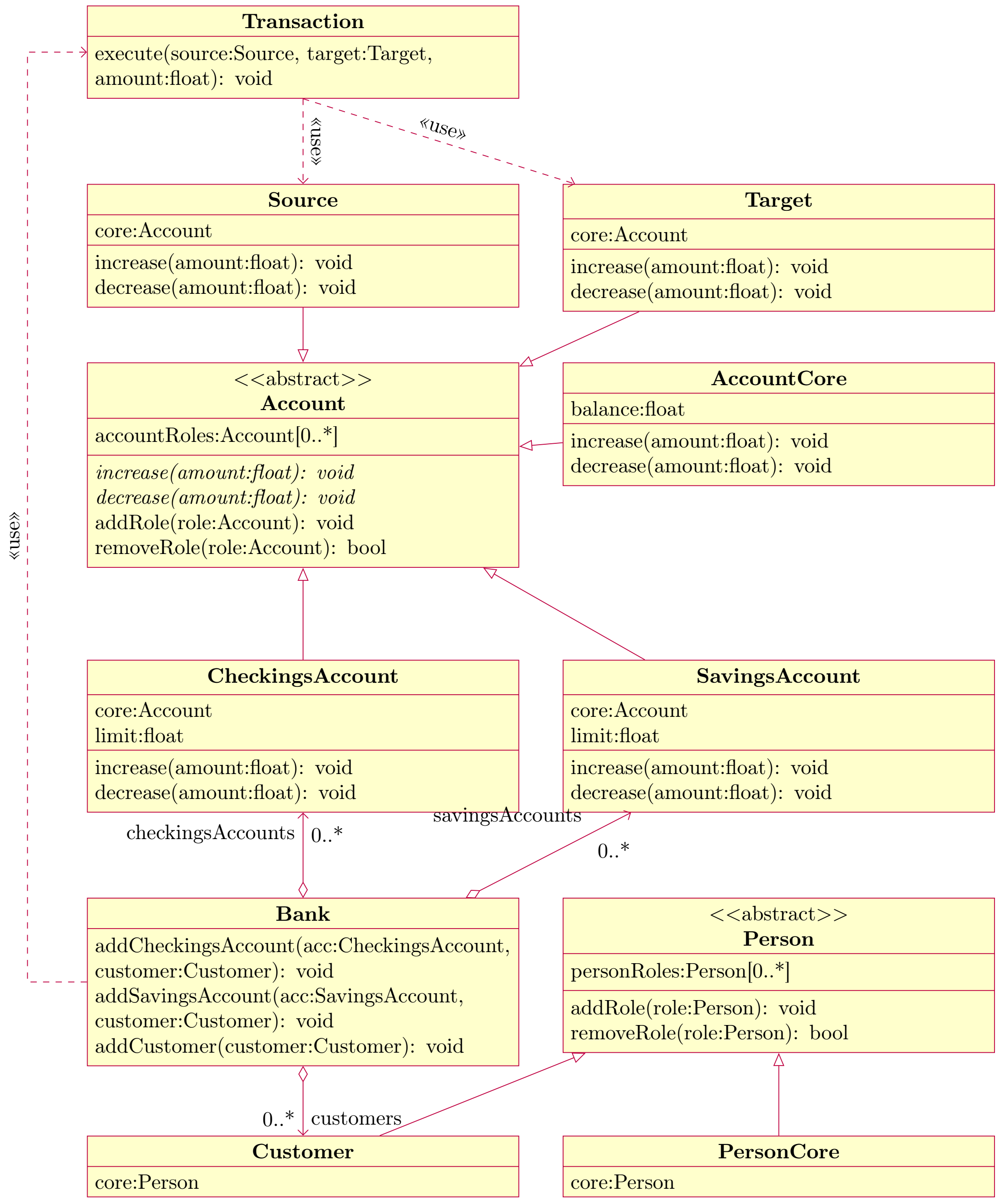
6.3.1 Role Object Pattern

Bäumer et al. have introduced the Role Object Pattern (ROP) because of the need for a flexible design pattern that allowed for unanticipated changes without the need to recompile the whole application [Bäu+97]. The pattern represents players and roles in an inheritance hierarchy. An abstract class of the entity represents the root, the player is split into a *core* class that implements the basic behavior. Role classes are inherited from the root and delegated to their core. It allows modeling of different views of an object designed as role objects which are dynamically added and removed from the core object. In Figure 6.3 the UML model of the role split pattern applied to the role model from Figure 6.2 is shown. The pattern has no realization for context-dependent roles. Thus, a `Bank` is represented as a normal class. Furthermore, as with many design patterns, the role object pattern suffers from *object schizophrenia* [Ken99], as common object-oriented programming languages do not support delegation, but a weaker form called *forwarding*. It is the programmer's responsibility to carefully implement the forwarding to the core object and to correctly override the method implementations resulting from the segmentation of the logical entity into multiple physical entities, where the identity of each entity is different. This responsibility applies to every single entity that should be able to play roles.

6.3.2 Object Teams/Java

Stephan Herrmann recognized that collaborations are a crosscutting concern as multiple classes are involved [Her03]. ObjectTeams/Java (OTJ)[14] is an approach to define crosscutting concerns realized as a combination of aspect-orientated programming and software composition. The programming language has already been introduced in Section 3.2.3 as a representation of roles embedded in a class-based object-oriented language. The implementation of a bank providing a `CheckingsAccount` based on OTJ callin mechanism is shown in Listing 6.12. The implementation of the benchmark using deep roles is shown in Listing 6.13. Object Teams [Her07] has been measured using *callin* to realize the source and target roles of the transaction.

[14]The website of OTJ: `http://www.eclipse.org/objectteams/`



Transaction
execute(source:Source, target:Target, amount:float): void
«use»
«use»
Source
core:Account
increase(amount:float): void
decrease(amount:float): void
Target
core:Account
increase(amount:float): void
decrease(amount:float): void
<<abstract>>
Account
accountRoles:Account[0..*]
increase(amount:float): void
decrease(amount:float): void
addRole(role:Account): void
removeRole(role:Account): bool
AccountCore
balance:float
increase(amount:float): void
decrease(amount:float): void
«use»
CheckingsAccount
core:Account
limit:float
increase(amount:float): void
decrease(amount:float): void
SavingsAccount
core:Account
limit:float
increase(amount:float): void
decrease(amount:float): void
savingsAccounts
checkingsAccounts
0..*
0..*
Bank
addCheckingsAccount(acc:CheckingsAccount, customer:Customer): void
addSavingsAccount(acc:SavingsAccount, customer:Customer): void
addCustomer(customer:Customer): void
<<abstract>>
Person
personRoles:Person[0..*]
addRole(role:Person): void
removeRole(role:Person): bool
0..*
customers
Customer
core:Person
PersonCore
core:Person

Figure 6.3: A UML class diagram representing the implementation of the role model from Figure 6.2 using the Role-Object-Pattern.

Listing 6.12 The implementation of a `Bank` which provides a `CheckingsAccount` based on OTJ's callin mechanism by replacing a base method.

```
team class Bank {
  class CheckingsAccount playedBy Account {
    callin withFee(float amount) {
    ...
    result = base.withFee(amount * FEE);
    ...
    return result;
    }
    withFee(float) <- replace withdraw(float)
  }
}
```

6.3.3 Role4J

Role4J[15], the successor of LyRT [Tai+16a], allows to vary the behavior of an application at the level of single objects. This adaptation mechanism is called *dynamic binding*. The implementation is provided as a Java library and offers a Java API to handle contextual adaptations and role-playing. However, to participate in role-playing the classes have to implement the `IPlayer`, `IRole`, and `ICompartment` interfaces, respectively. The runtime is designed to use a registry to handle all states in a central lookup table. The registry must be used to create new instances of players, roles, and compartments. It also stores the play relations between the players and the roles of the active compartment. Furthermore, it stores the level of the relation, e.g., when roles play roles (deep roles) the level will increase by one, and the sequence of the relation, i.e., the number of multiple roles bound at the same level. If multiple roles implement the same method, the method resolution mechanism chooses the one implemented by the role with the highest level and the sequence. At a given point in time, only one compartment can be active and every binding of roles to a core will be stored relative to the active compartment. The variability is achieved by generating subclasses at run-time. Into these subclasses, the dispatch logic is implemented using proxy instances. These can be exchanged without touching the core objects allowing for unanticipated adaption without restarting the application [Tai+16b]. Role4J uses ByteBuddy [Raf23] to realize the class rewriting.

[15]The repository of Role4J publicly available on GitHub: `https://github.com/nguonly/role4j`

Listing 6.13 ObjectTeams/Java implementation of the `BankBenchmark`.

```java
public class BankBenchmark extends Benchmark {

    private final Bank bank;

    @Override
    public boolean innerBenchmarkLoop(final int innerIterations) {
        bank.activate();
        float amount = 100.0f;
        for (Account from : bank.getCheckingAccounts()) {
            for (Account to : bank.getSavingAccounts()) {
                    CallinTransaction transaction = new CallinTransaction();
                    transaction.activate();
                    try {
                        transaction.execute(from, to, amount);
                    } catch (RuntimeException e) {
                        e.printStackTrace();
                    } finally {
                        transaction.deactivate();
                    }
            }
        }
        bank.deactivate();
        return true;
    }

    @Override
    public boolean setUp(final int innerIterations) {
        bank = new Bank();
        bank.activate();

        System.gc();

        for (int i = 0; i < innerIterations; ++i) {
            Person p = new Person();
            bank.addCustomer(p);
            Account sa = new Account(i, 1000.0f);
            Account ca = new Account(i, 1000.0f);
            bank.addSavingsAccount(p, sa);
            bank.addCheckingsAccount(p, ca);
        }
        return true;
    }
}
```

Listing 6.14 Role4J implementation of the `BankBenchmark`'s inner loop.

```
@Override
public boolean innerBenchmarkLoop(final int innerIterations) {
    this.bank.activate();
        float amount = 100.0f;
    for (Account from : bank.getCheckingAccounts()) {
        for (Account to : bank.getSavingAccounts()) {
            try {
                Transaction transaction = reg.newCore(Transaction.class);
                transaction.setAccounts(from, to);
                // binds Source and Target roles to from and to
                bank.executeTransaction(transaction, amount);
            } catch (RuntimeException e) {
                e.printStackTrace();
                return false;
            }
        }
    }
    this.bank.deactivate();
    return true;
}
```

Because of the limit of a single simultaneously active compartment, we implemented the transaction as another player and bound the `Target` and `Source` roles to the `Account`. Therefore, we could realize a working setting using a single compartment and deep roles. Because the implementation of Rol4J is already dated we had to compile and run it with Java 8 which will limit compareability. We show the main benchmark loop in Listing 6.14.

6.3.4 SCROLL

Recent implementations of roles in object-oriented programming languages required a specific runtime environment. Scala ROles Language (SCROLL)[16] is an internal DSL written in Scala that allows role-based programming without a specific runtime [LA15].

[16]The repository of SCROLL publicly available on GitHub: `https://github.com/max-leuthaeuser/SCROLL`

Listing 6.15 SCROLL implementation of the `BankBenchmark`'s inner loop.

```
override def innerBenchmarkLoop(innerIterations: Int): Boolean = {
  var amount: Float = 100.0f
  for (from <- bank.getCheckingAccounts()) {
    for (to <- bank.getSavingAccounts()) {
      val transaction = new Transaction
      transaction.rolePlaying.addPlaysRelation(from, new transaction.Source)
      transaction.rolePlaying.addPlaysRelation(to, new transaction.Target)
      transaction.compartmentRelations.partOf(bank)
      val transactionRole = new bank.TransactionRole
      bank.rolePlaying.addPlaysRelation(transaction, transactionRole)
      transactionRole.execute(amount)
    }
  }
  return true;
}
```

Conceptually, SCROLL uses a Single Underlying Model (SUM) and provides *Views* on that model. Roles are embedded in reified contexts, called *Compartments*. Activating a compartment activates all its contained roles. To view multiple compartments in a single view, the involved compartments need to be merged so their underlying DAGs are merged. Thus, a compartment mimics the behavior of a view. To each compartment, the role-playing state of its roles is stored in a DAG called the role-play graph. A role-playing object can be understood as a compound type, an intersection of the role types that the object is playing. However, the implementation does not use intersection types but requires technical aspects of Scala, such as the *dynamic marker trait* to mimic the concept of losing type safety. To regain some control, SCROLL provides a compiler plugin [Leu18] that compares a given CROM [Küh+14] model to validate role bindings in a SCROLL program w.r.t. that model. The dynamic trait uses compiler rewrites on a `Player` instance when a function is not available on the role-playing object (see Listing 3.7 for more detail). The program then queries the role-play graph to dispatch a function call to the valid implementation. The method dispatch can be configured to search the DAG by providing a *dispatch query* [Leu17a, p. 85]. The measured benchmark's inner loop implementation using SCROLL is shown in Listing 6.15.

6.4 Results

This section presents the results of the experiments we conducted according to the methodology described before. We will first present the results of the performance measurements and compare the approaches.

We selected the Role Object Pattern to serve as a baseline to compare the other approaches. We motivate this selection with the Role Object Pattern being the most lightweight approach to realizing roles in object-oriented programming based on inheritance and delegation to a core object. We think that language implementations should compile a program to the *most efficient* representation possible. However, as we will present later, implementations often use a single representation regardless of the features used, creating a *semantic gap* that results in an overhead in execution time and memory.

6.4.1 Performance and Scalability

Due to restrictions of the dependencies from Role4J, we were forced to use Java 1.8 to execute the benchmarks of Role4J and report the numbers of the other approaches using OpenJDK 20. Role4J could not complete larger problem sizes other than 10,000[17]. For this reason, we present the steady-state performance of the approaches with a problem size of 10,000 and a maximum heap size of 8GB separately in Figure 6.4. The figure presents the execution time in the log scale of each iteration of the first run while the second and third runs were not too much different.

As one can see, Role4J is the slowest approach being $4,428,827\times$ slower than the Role Object Pattern. While all other approaches reduced their execution time throughout the experiment, Rol4J's is increasing. The problem is that for each object that is adapted as a role, the Role4J runtime creates a new proxy class. This not only creates many more objects than necessary but also makes all profile-based optimizations from the JVM inapplicable as each role is marked as a newly observed class.

In Figure 6.5 we report the steady-state performance of problem sizes between 500,000 and 6,000,000 with a step size of 500,000 grouped by benchmark, heap size, and VM used. We see that in the experiments with deep roles and a heap size of 4G OTJ was aborted with problem sizes above 4,000,000 transactions due to the JVM reporting out-of-memory errors. Compared to the Role Object Pattern in the experiments with a heap size of 8G we find that the mean *slowdown* of OTJ is $54.28\times$ and SCROLL's mean slowdown is $123.0\times$. Across the different VMs we do not see striking

[17]We terminated the experiments of Rol4J after single runs not completing in 5 hours.

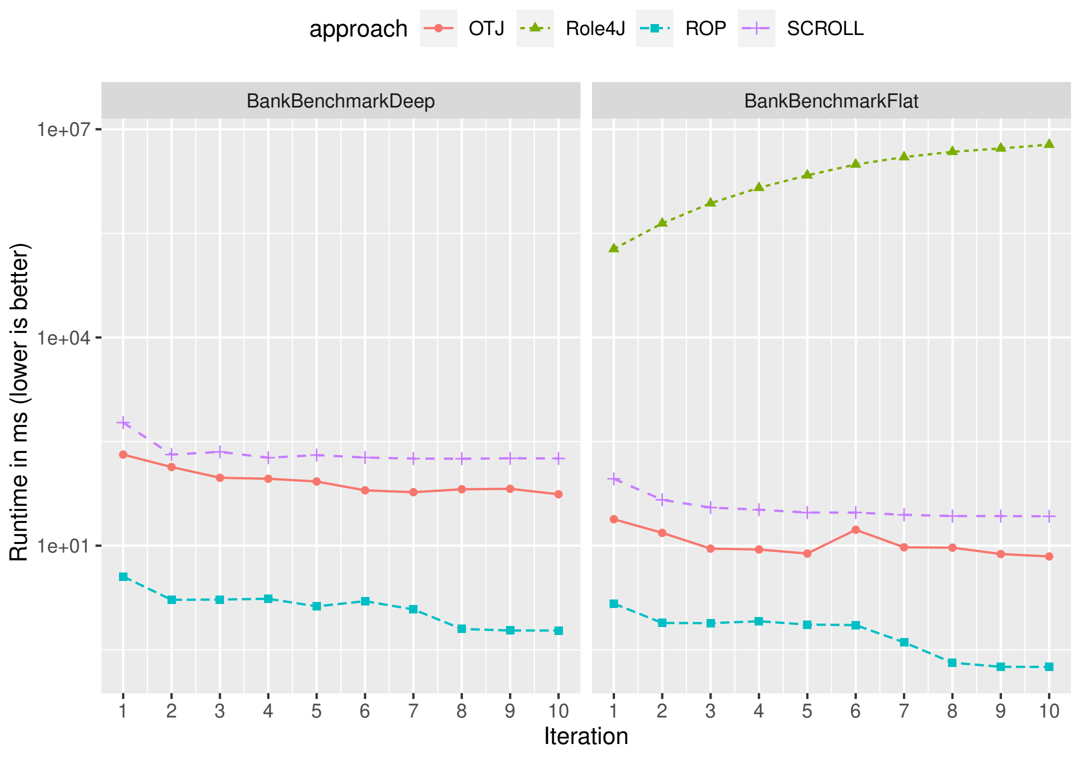


Figure 6.4: The execution time of each tested approach per iteration in the first run. Executed with OpenJDK 20 (OpenJDK 1.8 for Role4J, respectively) and a heap size of 8G. Results are shown for the benchmark using deep roles and flat roles.

differences, but the CoV for some reason is higher for the Role Object Pattern executed on OpenJ9 with a mean CoV of 1.50 (min 0.47, max 2.20) compared to the other VMs. OpenJ9 is a JVM developed with server applications in mind, thus we suspect the interference of the JIT compiler for the short-running application is introducing the variance. Compared to a prior comparison of the approaches [SC17] we find that a rewrite of SCROLL to support Scala 3 improved performance and scalability as it can handle a growing number of role-playing objects without the paralyzing overhead.

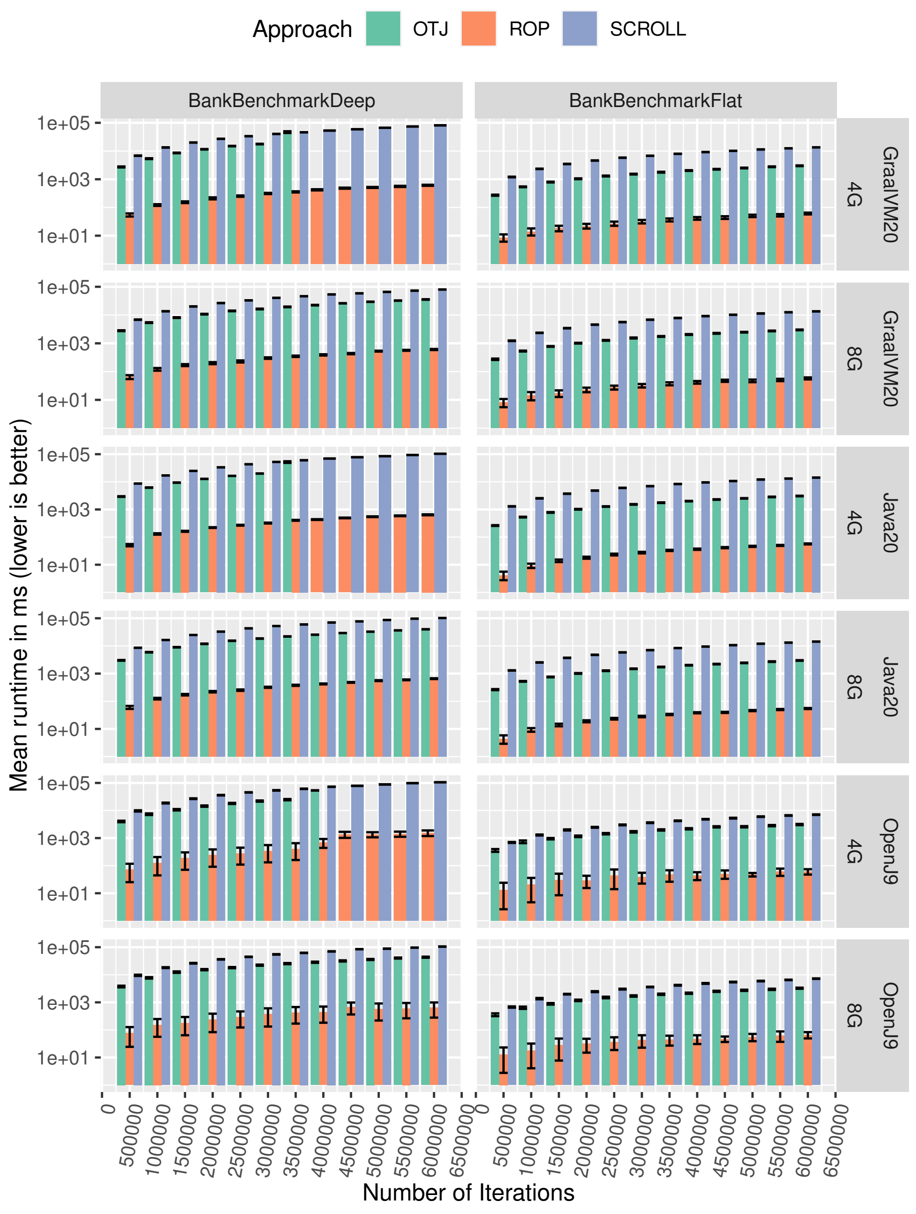


Figure 6.5: The mean of the execution time of the tested approaches among multiple JVM implementations, problem sizes, and benchmarks. Error bars show the standard error.

6.4.2 Time and Memory Contributions

The Role Object Pattern achieved the best results in all the approaches and all settings. We can see from Figure 6.5 that current approaches do not scale well. Even for small problems, the second fastest approach was at least 8.95× slower (SCROLL on OpenJ9, 4G heap size). While the approaches can leverage from JIT compilation they do not profit the same as the Role Object Pattern does. To better understand where time is spent, we monitored the JVM with VisualVM [Ora23c] when executing a benchmark with 2.25 million transactions. We do not only investigate the time spent in different parts of the applications but also take a look at what parts of the code are responsible for writing data to the heap. In the JVM a thread can acquire small regions of memory—called Thread Local Allocation Buffer (TLAB)—to allocate objects on the heap. When objects inside a TLAB die, i.e., are not life anymore, the garbage collector may free the memory region. The repeated allocation of many intermediate objects may result in more invocations of the garbage collector which may negatively impact the performance of an application. Figure 6.6 summarizes the regions grouped by their purpose. The upper graph shows how much of the overall heap space was acquired by a given group while the graph below shows the accumulated execution time of code that belongs to the groups.

Overall the approaches spent little time in the code regions of the benchmark harness. Most of the time was spent executing code regions that belong to the implementations of role dispatch and code that manages roles, e.g., the creation or lookup of existing roles. For example, in SCROLL most of the time during dispatch was spent visiting the DAG that stores the relation of naturals to their roles. In OTJ lots of the time was spent in generated dispatch methods. Improving in these areas will greatly improve the end-to-end latency of these approaches.

VisualVM reported that the Role Object Pattern and OTJ each used approximately 250MB heap memory to represent all types of accounts, transactions, and the bank itself. A huge difference can be seen in the cumulative acquisition of heap space. In this case, Role Object Pattern acquired an amount of 365MB of heap space, while OTJ acquired a total of 8.9GB. For 2,500 transactions SCROLL acquired 21.7GB and Role4J acquired 3.8GB. For 10,000 transactions Role4J acquired 16.4 GB of heap memory.

As explained before, Role4J stores all relations between the compartment, its players, and all roles in a central lookup table. When a new relation is added, its level and sequence are computed. This accounts for 89% of the execution time of the benchmark. That is because there is an exhaustive search over the central lookup table with an algorithmic complexity of $\mathcal{O}(N)$ with N being the number of stored relations, e.g., customers, accounts, and roles. Currently, the lookup table is implemented as an

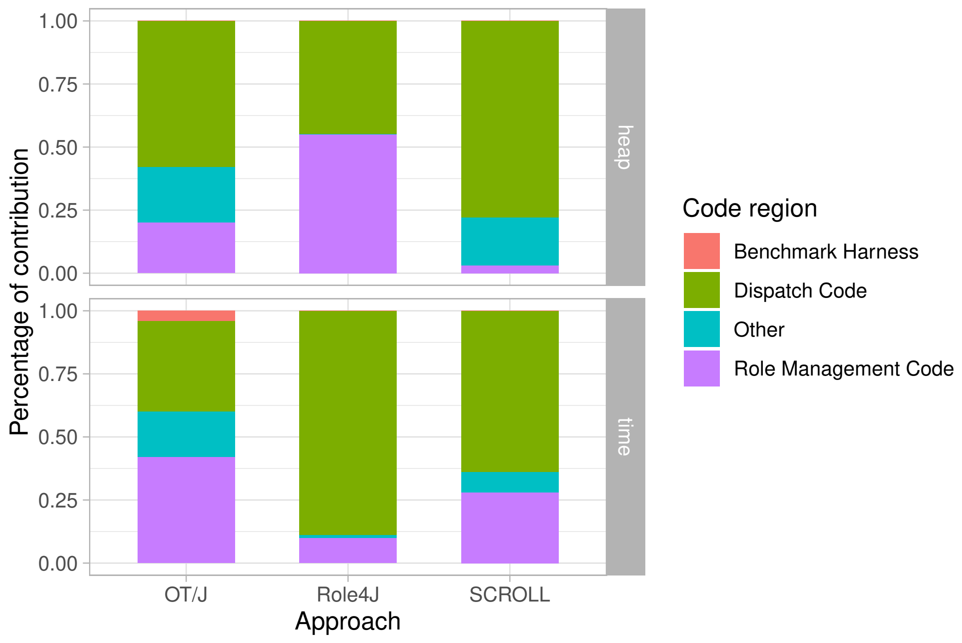


Figure 6.6: A stacked barplot of the deep roles benchmark size of 2,250,000. Top: Percentage of contribution to the heap of different code regions. Bottom: Time spent during the benchmark in these regions of code.

`ArrayDeque`[18] that stores the relations and is explored multiple times. Second, generating a new subclass for every player object and role object accounts for 26% of the overall heap pressure during the benchmark.

In ObjectTeams/Java, when an object begins to play a role in a team, the resulting role object is stored in a `WeakHashMap`. Elements in that collection can be garbage collected when free memory depletes. In the benchmark 17% of heap pressure (1.5 GB) is dedicated only to that cache structure. The implementation of callins in OTJ does extensive composition of strings at runtime, e.g., a method identifier consists of the name of the class defining the method and the type names of the parameters. These strings must be composed with every invocation of callins cumulatively acquiring 5.2GB of heap data, responsible for about 58% of overall pressure to the heap

Role-playing objects are realized with *compound objects* using Scala's `dynamic` trait.

[18] An `ArrayDeque` is a performant collection from the Java Collections Framework. This class is faster than LinkedList when used as a queue.

Using compound objects results in a high amount of boxing and unboxing instructions amounting to 64% of execution time spent during dispatching. SCROLL uses a DAG per compartment to map players to their played roles. In SCROLL, the dispatch requires a search in the DAG which accounts for 78% of GC pressure (16.9GB) as many intermediate objects are created. This flexibility for the method dispatch accounts for 64% of the execution time being spent in checking the equality of objects in the DAG.

6.5 Conclusion

Our results are in line with results from related approaches. In Lasagne/J [Tru+01] the disjunctive wrapper chaining is said to introduce a scalability problem due to the indirection of method execution through the dynamic wrapper chains. In the domain of AOP it is concluded that since object-oriented execution environments, i.e., virtual machines, do not understand aspect semantics, the aspect compiler produces a *verbose description of aspects* in an object-oriented paradigm which results in a high overhead [HS07]. The reason is that function invocations or property accesses are typical locations for join point shadows that will be decorated with residuals. Mechanisms for late binding of advices are not applied to such high-level concept, resulting in a semantic gap where residuals being evaluated each time in the worst case. This results in a severe performance penalty ranging from two orders of magnitude in AspectWerkz [Bon04] to performance losses of less than one decimal power [HM04] in Steamloom [Hau05].

7 The Semantic Gap of Roles in Object-Oriented Programs

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."
— Edsger Dijkstra

Throughout this thesis, we recurrently mentioned a *semantic gap* when describing Role-oriented Programming implementations. In Chapter 6 we closely benchmarked state-of-the-art approaches and measured a huge overhead in end-to-end latency. We concluded that it was caused by the extra overhead resulting from the implementation of advanced dispatch mechanisms provided by contextual roles, the management of contextual roles, and the creation of many intermediate objects increasing the overhead of additional garbage collection. This chapter takes a closer look into ObjectTeams/Java, a role-oriented programming language implementation providing most of the features ascribed to roles, and explains the observed overhead.

7.1 Compilation of ObjectTeams/Java Programs

The compilation of ObjectTeams/Java programs is divided into two phases. First, an OTJ program is statically compiled into Java Bytecode using the OTJ compiler, an extension of the Eclipse Java compiler. In this step, the compiler checks the static semantics of team and role definitions as well as binding declarations.[19] A class hierarchy analysis is used to identify possible inheritance hierarchies in team classes and whether callins override callins defined in superclasses. The compiler adds "magic methods" to each team class, for example, the method `callBefore` that implements the dispatch to all defined before callins of the team's encapsulated roles.[20] The compiler requires these analyses to statically map declarations of callins to identifiers as represented in

[19]Cf. Section 3.2.3 for an explanation of the terms used to describe OTJ semantics.

[20]A `team class` is realized as a normal Java class which happens to extend `Team` and implements the interface `ITeam`. The class `Team` already provides the mock implementations of these functions, the compiler only rewrites the function body accordingly.

Definition 7.1. A base call inside a replace callin, for example, `base.withFee` in the implementation of `SavingsAccount.withFee` as shown in Figure 6.2, will be replaced with the function call `this._OT$callNext`.

In this phase, the compiler only requires read access to base classes that are declared in the binding declarations. The semantic analysis does not require access to all subclasses of the declared base class. These will be processed in the second phase.

Definition 7.1 (Callin Identifier)**.** Given a team and a callin, the compiler assigns a new identifier. $callinId : Callin \times Team \rightarrow Int$ where $callinId(c, t) \mapsto \{n \mid \text{fresh}\, n \in \mathbb{N}\, \text{for}\, t\}$.

To take into account any subclass of declared base classes, the second phase in the compilation process is to *rewrite* all classes at runtime or load time, i.e. when classes get loaded into the JVM. The OTJ runtime provides a Java Virtual Machine Tooling Interface (JVMTI) interface where an agent can intercept class loading. This agent is used to transform the bytecodes of loaded classes and to rewrite methods declared in binding declarations.[21] For the rewrites, template-based code generation based on the ASM library [BLC02] is employed. For example, the original body of the `Account.decrease` function is moved into `Account.callOrig`. To identify the base method the compiler moved to `callOrig` an identifier is compiled that represents the base method. The identifier is created according to Definition 7.2 (see `boundMethodId` of the function signature in Listing 7.16). ObjectTeams/Java applies the envelope approach [Boc+05] to execute advices from join point shadows. The original body then is replaced by a local constant representing that identifier and a call to the generated function `callAllBindings`. The envelopes are the entry point to structured role dispatch and shown in Listing 7.17. To initialize role dispatch OTJ defines the envelope `callAllBindings`. The body of a base class `callAllBindings` always calls the OTJ runtime to get all active teams and callins. To realize the rewriting the runtime creates a MOP representing teams, base classes, and bindings.

Definition 7.2 (Bound Method Identifier)**.** Given a base method, the compiler assigns a fresh number that represents the base method. $boundMethodId : Method \times Base \rightarrow Int$ with $boundMethodId(m, b) \mapsto \{n \mid \text{fresh}\, n \in \mathbb{N}\, \text{for}\, b\}$.

[21] The Oracle JVM does not allow the redefinition of function signatures or introduction of new functions at runtime. Classes may get reloaded when a function body is rewritten. Thus, ObjectTeams/Java decided to prepare *all* classes as possible base classes in case a binding declaration referencing a base class is loaded after the base class itself has been loaded. We could imagine a completely different flow with a JVM that would not impose these restrictions, as shown in [WWS10].

Listing 7.16 The program code of the function `_OT$callNext` defined in class `Team` of the OTJ runtime. The implementation delegates to the next active team or to the original method if there is none.

```
public Object _OT$callNext(IBoundBase2 baze, ITeam[] teams, int idx,
    int[] callinIds, int boundMethodId, Object[] args,
    Object[] baseCallArgs, int baseCallFlags)
{
    // Are there still active teams?
    if (idx+1 < teams.length) {
        // Yes, so call the next team/callin
        return teams[idx+1]._OT$callAllBindings(baze, teams,
            idx+1, callinIds, boundMethodId, args);
    } else {
        //No, call the base method
        return baze._OT$callOrig(boundMethodId, args);
    }
}
```

7.2 The Meta-Object Protocol of ObjectTeams/Java

In Chapter 1 we introduced the meta-hierarchy [FD99] and meta-objects that represent abstractions of objects on a higher level. For reflective systems, it can be stated that "*[t]he metaobjects on the metalevel describe the structure, the features, and the semantics of the domain objects.*" [Aßm03] The *open implementation* [Kic96] design principle proposes to expose a part of the implementation strategy to the application level. This principle allows the design of systems to be open for later (unanticipated) adaptations. Metaobject Protocols (MOPs) allows the definition of internal DSLs that extend the capabilities of a given host language using the host language itself [KDB91]. In particular, a meta-object protocol can be seen as an open implementation of an object-oriented language. Both reflection and MOPs has been used to implement AOP technology by adapting the semantic properties of the aspect language [DMB09], applying crosscutting concerns with Composition Filters [BA01; BA04], and using the meta-object capabilities of the host language Squeak in AspectS [Hir01; Hir03] to name a view.

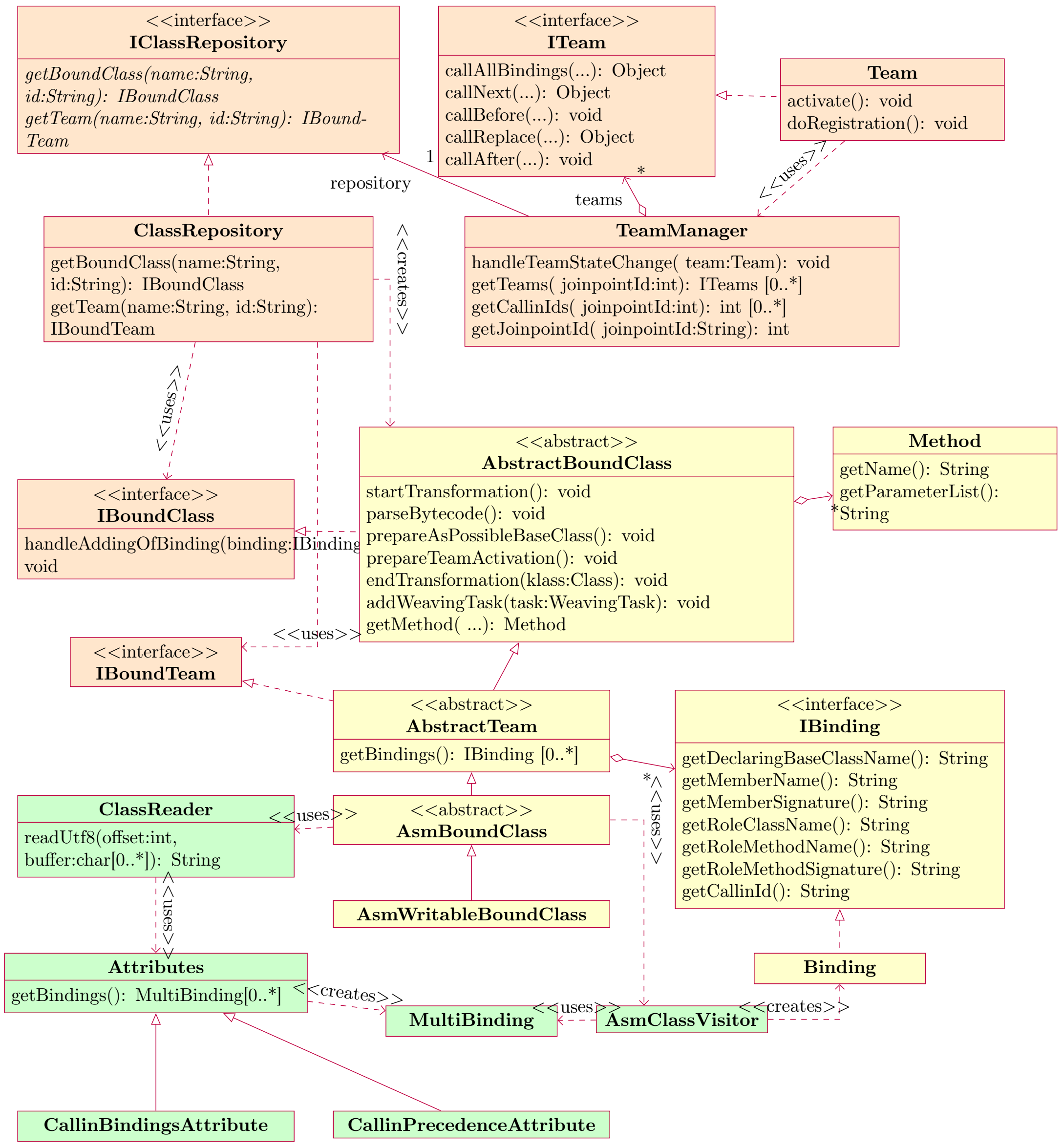
In the domain of role-oriented programming languages, SCROLL uses a Metaobject Protocol [Leu17a, Ch. 9] to represent the semantics of contextual roles. While it integrates well with the Scala ecosystem it incurs an execution overhead as shown in prior works [SC17] and Chapter 6. In ObjectTeams/Java the implementation of contextual roles is based on a combination of meta objects used at runtime and a (runtime) code generator that uses the MOP to encode the semantics of role dispatch in Java Bytecode. Furthermore, the Metaobject Protocol encodes facts about the extended type system provided by ObjectTeams/Java. [22] To increase the performance of the OTJ runtime, a pure reflective approach (more on this later) is not used to realize the extended semantics of contextual roles. Instead, as presented throughout Chapter 6, the decision-making is already included in the generated code, which is partly compiled from the MOP at runtime.

In Figure 7.1 we present a UML class diagram of the MOP implemented in OTJ. To keep the model approachable we only show the immediately necessary parts. Furthermore, we use color coding to assign classes to different groups highlighting their purpose. Classes colored orange are responsible for the execution and management of base classes, roles, and teams. They constitute the parts that define the OTJ runtime. A yellow color means a class is a metaclass describing domain elements, i.e., (callin) bindings, base classes, and teams. They provide the metadata to the runtime. The classes in green are responsible to read and return the appropriate metaclasses extracted from the metadata stored in the class file format.

Every class that is loaded at runtime is registered with the `ClassRepository` and is assigned one of the metaclasses: `AbstractBoundClass` captures that a class is a (possible) base class and `AbstractTeam` captures that a class is a team. The interfaces `IBoundClass` and `IBoundTeam` abstract implementation details the OTJ runtime does not care about. Roles do not have a direct representation in the model but are subsumed by `Binding` which stores the role class name, the role method name, and the role method signature. This is acceptable since the role class was already compiled by the Object Teams compiler as an inner class of Teams and the code to dispatch to the role is already generated.

The `TeamManager` is notified for each activation and deactivation of a `Team` instance. This triggers the handling of the state change. The `TeamManager` will query the `ClassRepository` for the metaobject `AbstractTeam` of the team instance to collect the provided bindings. For each base class that is referenced therein the corresponding `IBoundClass` will be queried and queued for rewriting if that has not happened already. Th The `TeamManager` is responsible for bookkeeping active teams and their provided bindings. This data will be requested whenever the method `callAllBindings` prepares

[22]The phenomenon in OT that from a typing perspective, the type of two roles from two different team instances are unequal has been coined *family polymorphism* [Ern01].



<<interface>>
IClassRepository
getBoundClass(name:String, id:String): IBoundClass
getTeam(name:String, id:String): IBoundTeam
<<interface>>
ITeam
callAllBindings(...): Object
callNext(...): Object
callBefore(...): void
callReplace(...): Object
callAfter(...): void
Team
activate(): void
doRegistration(): void
1
repository
*
teams
<<uses>>
ClassRepository
getBoundClass(name:String, id:String): IBoundClass
getTeam(name:String, id:String): IBoundTeam
<<creates>>
TeamManager
handleTeamStateChange(team:Team): void
getTeams(joinpointId:int): ITeams [0..*]
getCallinIds(joinpointId:int): int [0..*]
getJoinpointId(joinpointId:String): int
<<uses>>
<<abstract>>
AbstractBoundClass
startTransformation(): void
parseBytecode(): void
prepareAsPossibleBaseClass(): void
prepareTeamActivation(): void
endTransformation(klass:Class): void
addWeavingTask(task:WeavingTask): void
getMethod(...): Method
Method
getName(): String
getParameterList(): String
*
<<interface>>
IBoundClass
handleAddingOfBinding(binding:IBinding): void
<<uses>>
<<interface>>
IBoundTeam
<<abstract>>
AbstractTeam
getBindings(): IBinding [0..*]
<<interface>>
IBinding
getDeclaringBaseClassName(): String
getMemberName(): String
getMemberSignature(): String
getRoleClassName(): String
getRoleMethodName(): String
getRoleMethodSignature(): String
getCallinId(): String
*
<<uses>>
ClassReader
readUtf8(offset:int, buffer:char[0..*]): String
<<uses>>
<<abstract>>
AsmBoundClass
AsmWritableBoundClass
<<uses>>
Attributes
getBindings(): MultiBinding[0..*]
<<creates>>
MultiBinding
<<uses>>
AsmClassVisitor
<<creates>>
Binding
CallinBindingsAttribute
CallinPrecedenceAttribute

Figure 7.1: A UML class diagram presenting an excerpt of the implementation of the Metaobject Protocol (MOP) from ObjectTeams/Java. Yellow-colored classes account for the representation of contextual roles, orange-colored classes account for the execution and management of the MOP, and green-colored classes account for the serialization of OTJ-specific metadata in the class files.

Listing 7.17 The Java interfaces of the envelope methods `callAllBindings` and `callNext` in ObjectTeams/Java.

```
// The Java interface of callAllBindings
Object callAllBindings(IBoundBase base, Team[] teams, int index,
  int[] callinIds, int boundMethodId, Object[] originalArguments);

// The Java interface of callNext
Object callNext(IBoundBase base, Team[] teams, int index,
  int[] callinIds, int boundMethodId, Object[] originalArguments,
  Object[] arguments, int superCall);
```

a role dispatch.

7.3 Method Dispatch in ObjectTeams/Java

Now that we know how an OTJ program is produced let us discuss how a role method is looked up and how method dispatch is implemented. This equips us with the prerequisite to understand the overhead found in our quantitative analysis in Chapter 6 and will allow us to provide a comprehensive and compelling explanation of the results. Contemporary object-oriented programming languages implement the lookup and execution of a call target as two distinct processes (with distinct possible optimizations). Given the invocation of a method on a polymorphic reference, late binding ensures that the correct method is looked up and subsequently executed. In OTJ, however, the process of lookup and dispatch is intermingled and thus, must be explained together. As a result of embedding the role semantics in the Java Bytecode, there exists no higher-level notion of a role-polymorphic reference and no mechanism for late binding that respects the semantics of contexts and roles. To overcome these challenges the language is implemented using a template-based compilation scheme where the compiled code contains dispatch code for each possible case. Thus, dispatch code is part of the application which also affects optimizations employed by the JIT compiler on the generated code. At this level, there is no possibility to account for special cases such as turning a virtual function call into a direct call when it is known that there is no subclass loaded into the JVM. As a consequence, the scheme used to implement role dispatch does not allow for *late binding* which normally is used to bind the results of lookups into a polymorphic call site.

To get an overall picture and to explain the different core parts in this process we will

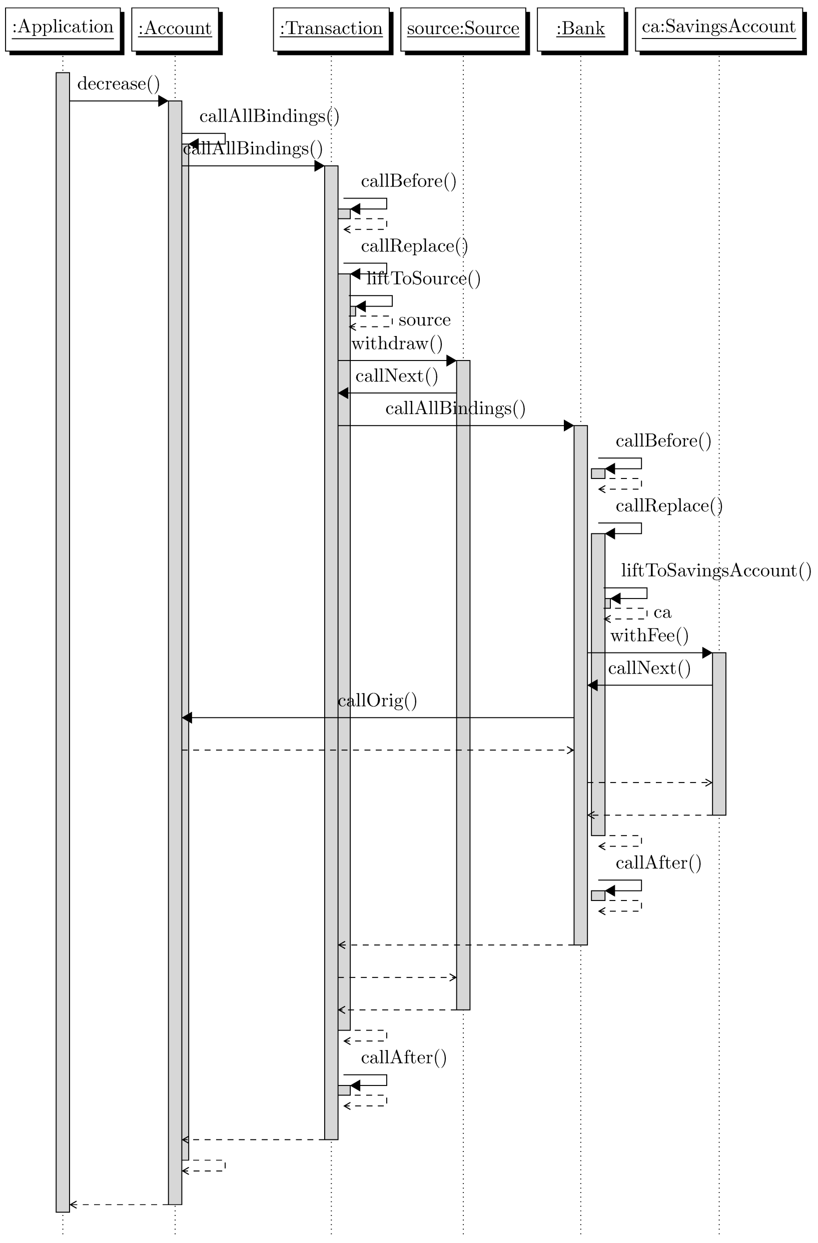


Figure 7.2: A simplified UML sequence diagram showing the resolution of the function call `Account.decrease()` with active compartments `Transaction` and `Bank`. For readability we omitted leading "`_OT$`" of functions belonging to the OTJ runtime.

refer to the UML sequence diagram in Figure 7.2 which shows the steps of resolving the dispatch of a call to `Account.decrease()` when two teams that declare a binding for that method are active. The body of `Account.decrease()` had been rewritten at load time to delegate to `callAllBindings` because there are binding declarations in roles that define callins for the `decrease` function of `Account`. As we can see in Listing 7.17 the interfaces of `callAllBindings` and `callNext` capture the relevant runtime information to execute role dispatch. The function `callAllBindings` of the next active team, where precedence order is defined on the activation order of teams, is called next. This function implements the dispatch to before callins, to replace callins, and finally to after callins. While before and after callins can only add behavior, the replace callin can change behavior. The base call in `Source.withdraw()` invokes `callNext`. The method—which is shown in Listing 7.16—is central to looking up more role methods; either it delegates to the next active team's `callAllBindings` or the original method is called. The function signature exposes all the runtime data that is passed around to realize role dispatch. A copy of the runtime's stack of active teams and their callin identifiers is carried freeing the runtime to provide measures to protect the values from being altered. We can see in Figure 7.2 that dispatch in the next team follows the same structure and the whole process is repeated.

7.4 The Overhead of Role Dispatch

Because OTJ, as well as other role-oriented programming languages, *simulate* contextual roles in the class-based object-oriented programming languages the role language runtime must orchestrate the role dispatch. Contemporary programming languages implement the lookup and execution of a call target as two distinct processes (with distinct possible optimizations). The possibilities to optimize, to name a view, range from capturing different implementations of a same-named function into a call site, called Polymorphic Inline Cache (PIC) [HCU91] or to turn a virtual function call into a direct call when it is known[23] that there is no subclass loaded into the JVM.

The role-oriented programming languages we have reviewed intermingle the process of lookup and dispatch. In these implementations, the two processes are indistinguishable from the rest of the application and are supported by ordinary host-level data structures. For example, OTJ [Her03] uses lists and arrays containing active team instances and active callins, SCROLL [LA15] uses a DAG to connect role instances, and Role4J [Tai+16a] uses a linked list to store relations between instances.

[23]The knowledge is inferred at runtime and may be falsified. This kind of optimistic assumption requires the VM to provide deoptimization capabilities.

All approaches have in common that dispatch is realized as a (predefined) traversal over the content of these structures. They also have in common to not take into account the potential for optimization depending on application states. The implementations entrust all runtime optimizations to the JIT compiler of the VM. The result is that the benefit of optimizations tuned to optimize common Java programs does not transfer to optimizations applied to role-oriented, Java-based programs. On the contrary, the extra indirections can defeat attempts by the JVM to predict and inline call targets.

Part III

Runtime Optimization with Partial Evaluation

8 Optimizing Role Dispatch with Partial Evaluation of Meta-Objects

"The whole is greater than the sum of the parts."

— Aristotle

The semantic gap of *simulating* role-oriented concepts in an object-oriented target machine model has been discussed in Chapter 7. This semantic gap has also been faced in aspect-oriented code between the language's expressions and their realization [HS07]. Haupt et al. proposed to regard join points, i.e., points in the execution flow of a program, as a locus of *late binding*. Other approaches focused on language support to allow the superimposition of expressions (methods) on method calls, i.e., the principles of advice in AOP, using method-call interception [Läm02]. The application of *partial evaluation* [JGS94] to gain efficiency in the context of MOPs [MY98] was also discussed. The strength of partial evaluation has also been shown in the context of the compilation process of AOP programs in removing unnecessary run-time checks [MKD03]. These works have not been ported to contemporary AOP, COP, nor role-oriented programming language implementations. However, in recent years interest in PE spiked as an efficient means to optimize JIT compiled programs [Sea16].

A big research challenge up to date is that the proposed formal models only consider the *static* binding of advice, spoken in terms of AOP. There is a long way to practically meaningful language design covering the essentials of AOP implementations like AspectJ [Kic+01], COP implementations like JCop [AH12b], or role-oriented programming languages like OTJ [Her07].

In this chapter, we consider a new implementation model of role dispatch in the role-oriented programming language ObjectTeams/Java inspired by Partial Evaluation. The current execution model intermingles the process of lookup and dispatch in sequences of instructions. We will repurpose the MOP used to transform, lookup, and execute OTJ programs to allow separate processes for lookup and dispatch. Inspired by PE we propose to evaluate the runtime state to enable computing the call targets according to the evaluated state. The execution of calls at join points will be intercepted and the returned composition of calls will be installed. As a result, the implementation can reuse lookups and enable the late binding of role dispatch at join points. To further speed up dynamic programs we extend the notion of polymorphism to contextual roles and implement Polymorphic Inline Caches (PICs) at join points.

8.1 Reference Object Model

Formal model-based solutions such as CROM [Küh+14] and CROI [Küh+15] miss important semantic details such as concrete bindings and method signatures, thus, are not a well-suited abstraction to represent a programming language. There exist type systems for role-based languages with contexts such as Featherweight EpsilonJ (FEJ) [KT10], or Method Call Interception (MCI) [Läm02] for AOP-like abstractions. But they omit the intricacies of supporting more complex binding mechanisms such as bindings with replace callins and base calls (`around` with `proceed` in AOP) provided by OTJ. The possible rewrites of the control flow require a special handling of these join points. Judging over these (complex) binding types is required to overcome the semantic gap.

The set of ObjectTeams types $\mathcal{T}$ is defined as the union of team types $\mathcal{C}$, role types $\mathcal{R}$, and base types $\mathcal{B}$. We do not consider modules, packages, or namespaces but flat collections of types. A nice property of this approach is that in the cases where a context type fills a role type, we have $\{\exists C | C \in \mathcal{B} \wedge C \in \mathcal{C}\}$. Also deep roles, i.e., role types fill role types, can be represented as $\{\exists C | C \in \mathcal{B} \wedge C \in \mathcal{R}\}$.

To represent instances we use lowercase letters. The model also allows to judge over types of instances such as the reflective Java function `instanceof(c)` returns an instance of `Class<?>`; the runtime representation of the class of `c`. The type of an instance c is defined as the meta-function $isa(c) = T$ with $T \in \mathcal{T}$.

Given a class $C \in \mathcal{T}$, $M(C)$ denotes the set of methods of that class. The signature of method m in class C is denoted as $sign(m, C) = T_1 \times \cdots \times T_n \rightarrow T_r$ for $m \in M(C)$.

Definition 8.1 (Method Applicability)**.** Given a class $C \in \mathcal{T}$ and a method implementation $m \in M(C)$ we define $sign(m, C) = T_1 \times \cdots \times T_n \rightarrow T_r$. We can apply the method m to a method invocation $o.m(v_1, \cdots, v_n)$ when $isa(o) = C$ and $\forall i, 1 \leq i \leq n, isa(v_i) = T_i$.

Furthermore, $CB(T)$ denotes the set of callin bindings of the class $T \in \mathcal{C}$. A callin binding $cb \in CB(T)$ is a tuple (R, B, m_R, m_B) where $R \in \mathcal{R}$ is the role type that defines the binding, $B \in \mathcal{B}$ is the base class that is bound, $m_R \in M(R)$ is the role method and $m_B \in M(B)$ is the base method that is bound. The domain of a callin binding $dom(cb, T)$ for $cb \in CB(T)$ denotes the domain of the binding, which is either of the callin value types (CVT) where $CVT = \{before, after, replace\}$. The meta-function $loc(cb, T)$ denotes the line number where cb is defined in T which is used to disambiguate bindings in the same team with the same domain.

The evaluation of join points with active bindings requires that we assign a precedence order to active bindings. Therefore, a relation $\leq_{CVT} \subseteq CB(T) \times CB(T)$ where $T \in \mathcal{C}$ is defined as follows.

Definition 8.2 (Local Order of Callin Bindings)**.** Given two callin bindings $cb_1, cb_2 \in CB(T)$, cb_2 has lower precedence than cb_1 (denoted as $cb_2 \leq_{CVT} cb_1$) iff one of the following condition holds:

- $dom(cb_1) = \{before\}$ and $dom(cb_2) \in \{after, replace\}$,
- $dom(cb_1) = \{after\}$ and $dom(cb_2) = \{replace\}$,
- $dom(cb_1) = dom(cb_2)$ and $loc(cb_2, T) < loc(cb_1)$.

At a given point in time, an OTJ program may have multiple *active* team instances represented as the tuple $\gamma = (c_0, \cdots, c_{n-1})$. γ is said to represent the runtime state of the application. Let $\pi_k(\gamma) = c_k$ be the k-th entry of γ. The components in γ with a lower index represent the *recently activated* team instances, i.e., the tuples are ordered ascending by the duration the respective team instances are active. We denote $\prod_l^k(\gamma) = (\pi_k(\gamma), \cdots, \pi_l(\gamma))$ as the projection of γ into a runtime state representing the $l - k$ contexts in γ starting from k.

Definition 8.3 (Runtime State)**.** Given the n-tuple $\gamma = (c_0, \cdots, c_{n-1})$ and $\forall i, 0 \leq i \leq n-1$ s.t. $\pi_i(\gamma) = c_i$ it holds that $isa(c_i) \in \mathcal{C}$.

The application of behavior adaptations is defined concerning the order team instances appear in γ. Therefore, a relation $\preceq_\gamma \subseteq CB(T) \times CB(T)$ on the callin bindings provided by the active team instances in γ is defined as follows.

Definition 8.4 (Total Order of active Callin Bindings)**.** Given j, k where $0 \leq j \leq k \leq n-1$ and $\gamma = (c_0, \ldots, c_{n-1})$. Two team instances $\pi_j(\gamma) = c_j$, $\pi_k(\gamma) = c_k$ such that $isa(c_j) = T_j$, $isa(c_k) = T_k$. We define precedence for all callin bindings $cb_k \in CB(T_k)$ and for all $cb_j \in CB(T_j)$ saying cb_k has lower precedence than cb_j (denoted as $CB(T_k) \preceq_\gamma CB(T_j)$) iff one of the following condition holds:

- $j < k$,
- $cb_k \leq_{CVT} cb_j$ iff $j = k$.

Listing 8.18 The OTJ implementation of the `callAllBindings.java` function that delegates to other teams.

```
public Object _OT$callAllBindings(IBoundBase2 baze, ITeam[] teams, int idx,
    int[] callinIds,int boundMethodId, Object[] args)
{
    Object res = null;
    this._OT$callBefore(baze, callinIds[idx], boundMethodId, args);
    res = this._OT$callReplace(baze, teams, idx, callinIds,
        boundMethodId, args);
    // make result available to param mappings!
    this._OT$callAfter(baze, callinIds[idx], boundMethodId, args, res);
    return res;
}
```

8.2 The Case for Partial Evaluation in Role Dispatch

Let us draw our attention to the program code that encodes the role dispatch semantics. Chapter 6 already presented a detailed account of the implementation of role dispatch in OTJ. However, we never introduced the evaluation strategy employed at runtime that leads to the sequence of method calls shown in Figure 7.2. That is, there is a particular evaluation strategy compiled into the code that represents the dynamic semantics of role dispatch. This section introduces that evaluation strategy and draws a connection between the evaluation of role dispatch at runtime and runtime-generated dispatch code. We show that by partially evaluating the state of the runtime plus inferring the call targets using the MOP of OTJ the generated dispatch methods become unnecessary.

Let us take a look at Listing 8.18. Imagine, at the time of invocation, we observe the data values

$$\texttt{teams} \leftarrow \texttt{[bank]}, \texttt{idx} \leftarrow \texttt{0}, \texttt{callinIds} \leftarrow \texttt{[0]}.$$

The array `teams` is the implementation of the object model element $\gamma = (bank)$. For efficiency, the runtime reads the binding attributes once and stores them for subsequent access. The array `callinIds` holds all identifiers of callins provided by the respective team instance at the same index. Thus, when a team contributes multiple callin bindings the array `teams` repeats the same team instance for each binding. The reason is easier access to the arrays coordinated by the index variable `idx`. For each binding in `callinIds` one of the methods that delegate to role methods—`callBefore`, `callAfter`, or `callReplace`—will have a catch clause for that callin identifier. The components are ordered according to $\preceq_\gamma$.

$$
\begin{aligned}
&[\![\text{bank.callAllBindings}]\!][acc, [bank], 0, [0], 0, 100] \\
&\equiv [\![\text{bank.callBefore}]\!][acc, [bank], 0, [0], 0, 100] \\
&\circ [\![\text{bank.callReplace}]\!][acc, [bank], 0, [0], 0, 100] \\
&\circ [\![\text{bank.callAfter}]\!][acc, [bank], 0, [0], 0, 100] \\
&\equiv [\![\text{skip}]\!][acc, [bank], 0, [0], 0, 100] \\
&\circ [\![\text{switch (callinIds[idx]) case } 0 : \ldots]\!][acc, [bank], 0, [0], 0, 100] \\
&\circ [\![\text{skip}]\!][acc, [bank], 0, [0], 0, 100] \\
&\equiv [\![\text{SavingsAccount sa} = \text{bank.liftTo...}]\!][acc, [bank], 0, [0], 0, 100]
\end{aligned}
\tag{8.1}
$$

Given the join point (e.g., `callAllBindings`) of a ObjectTeams/Java program and the data values enumerated earlier applied to the join point. By partially evaluating the program using the data values we can partially pre-compute the dispatch of that program as demonstrated in Equation 8.1. The partial evaluator must know the point it has to stop, otherwise the unimportant code will be included in the partial evaluation and the compiled code will become too large [Wür+17]. For example, due to the partial evaluation, it becomes apparent that `callBefore` and `callAfter` in this example do not contribute any role methods and thus can be dead code eliminated from the program (denoted as $[\![\text{skip}]\!]$). If there would be a contribution of `callBefore` and `callAfter` we could have them partially evaluated as we did with `callReplace`. We also see the limit of this approach as we cannot partially evaluate the lifting because the result of that function requires the execution of the whole program.

We gave examples of how the approach of partially evaluating role dispatch is beneficial because we can pre-compute parts of the role dispatch, thus, reducing the overhead introduced by dispatch code of ObjectTeams/Java. However, this would require the implementation of a partial evaluator $[\![\_]\!]$ that can evaluate a Java program at runtime, i.e., requires physical access to methods such as `callAllBindings`, access to runtime values, and the capabilities to deploying the resulting program code into the Virtual Machine. The OTJ Metaobject Protocol as presented in Figure 7.1 is delivering all the required properties. It gives access to classes and their methods, as well as possibilities to alter them. The runtime provides the data values we require for partial evaluation. However, we encounter the issue that, for instance, `callAllBindings` is a method central to dispatching role methods and is called by each join point. Thus, the

Listing 8.19 The OTJ implementation of the `callReplace` function that lifts base classes to roles and implements calls to role functions. We used Java to present an excerpt of the originally generated Java Bytecodes. `this` is the `Team` instance stored in `teams[idx]`.

```
public Object _OT$callReplace(IBoundBase2 baze, ITeam[] teams,
    int idx, int[] callinIds, int boundMethodId, Object[] args)
{
  switch (callinIds[idx]) {
    case 0:
      Account acc = (Account) baze;
      Bank$SavingsAccount sa = this._OT$liftTo$SavingsAccount(acc);
      float amount = args[0];
      return sa.limited(baze, teams, idx, callinIds,
        boundMethodId, args, amount);

    case 1:
        ...
    default:
      return _OT$callNext(baze, teams, idx, callinIds,
        boundMethodId, args, null, 0);
  }
}
```

partial evaluator is tasked to create a specialization of the dispatch code for each join point. The resulting specialized code would have to be installed at the same program location. Moreover, the data values used to partially evaluate the dispatch code are not stable, thus on change, the code must be invalidated and rewritten or rolled back. This would incur a high overhead in time and memory requirements.

8.3 Partial Evaluation of Runtime State using Meta-Objects

The MOP of OTJ captures all information to infer the required properties of the role-oriented program under execution. At present a weaver generates dispatch code at "*load time*" by evaluating the Metaobject Protocol of OTJ. As presented before, this code is responsible for evaluating the runtime information and for subsequently executing the related role functions. However, if one could delay code generation until the execution of a function bound by a role, one could create a dispatch code that only executes the necessary minimum of functions. In such a case the information that is needed to compute this necessary minimum is the active team instances γ and the respective bindings $CB(T)$ for γ. The runtime stores active context instances $\gamma = (c_0, \ldots, c_{n-1})$ of length $n \geq 0$. Entries in γ are ordered so recently activated contexts come first. This reflects that recently activated contexts provide roles with a higher priority than older contexts. From that information, a *dispatch plan* can be computed that is subsequently executed.

8.3.1 Inferring Dispatch Plans from Runtime State

The `TeamManager` functions as a central point of truth for the OTJ runtime. As shown in Figure 7.1 the current implementation of OTJ stores the state of the application w.r.t. active bindings and active teams in the `TeamManager`, i.e., all required information to implement role dispatch. We propose *Dispatch Plans* that are a schedule of role functions for a specific call site (i.e., base method) and runtime state. The approach supports the open-world assumption of lazily loaded types at run-time and is inspired by Partial Evaluation (PE). We show that by partially evaluating the state of the runtime plus inferring the call targets using the MOP of OTJ the originally generated dispatch methods such as `callAllBindings`, `callBefore` etc. become unnecessary.

Algorithm 1 presents how to partially evaluate the runtime state of the application to construct a schedule of role methods. The static input to the algorithm is the base method whose behavior has to be changed, all active context instances, and an index representing how many context instances already had been evaluated. At the beginning of a role dispatch (see `callAllBindings` in Figure 7.2) the index is 0, otherwise $0 \leq i \leq n-1$. Given the static input, a schedule can be constructed. The algorithm does not execute scheduled elements as the schedule still contains unbound variables. These represent the dynamic input such as the concrete instance of the base class and the values applied to the function invocation. Information and meta-information required to construct the schedule are provided by the MOP of ObjectTeams/Java.

The algorithm infers all before, after, and if necessary, replace callins. First, in lines 1–5 it initializes the variables that hold intermediate results. For example, each type of callin is sequentially composed into a group with same-kinded callins. When the overall schedule is created the groups are sequentially composed. The sequential composition $m_1; m_2$ represents the imperative sequence of functions such that m_1 is executed before m_2. The initialization of the before and after callins is the identity function because the sequential composition holds that $id; m \equiv m$. The replace callin

Algorithm 1 Construction of a Dispatch Plan

In: Function m declared in base class $B \in \mathcal{B}$ with $sign(m, B) = T_1 \times \cdots \times T_n \rightarrow T_r$
In: Active context instances $\gamma = (c_0, \ldots, c_{n-1})$ and the current index i
Out: Dispatch plan $\mathcal{P}$ as a sequential composition ; of role functions

```
1: function ComposePlan($\gamma, m, i$)
2:     $after \leftarrow id$                                  $\triangleright$ $id$ is the identity function
3:     $before \leftarrow id$
4:     $replace \leftarrow m$
5:     $proceed \leftarrow true$
6:     while $proceed$ do
7:         $c_i \leftarrow \pi_i(\gamma)$                          $\triangleright$ Access $i$-th active context
8:         $T \leftarrow isa(c_i)$
9:         for $cb \leftarrow CB(T)$ do
10:            $m_B \leftarrow \pi_2(cb)$                       $\triangleright$ $cb = (R, B, m_R, m_B)$
11:            if $m_B = m$ then
12:                $lifted \leftarrow parmap_{cb}(m)$          $\triangleright$ Lift signature of $m$ to $m_R$
13:                if $dom(cb, T) = \{before\}$ then
14:                    $before \leftarrow before; lifted$
15:                else if $dom(cb, T) = \{after\}$ then
16:                    $after \leftarrow after; lifted$
17:                else if $dom(cb, T) = \{replace\}$ then
18:                    $replace \leftarrow lifted$
19:                    $proceed \leftarrow false$
20:                end if
21:            end if
22:        end for
23:        $i \leftarrow i + 1$
24:    end while
25:    return $before; replace; after$
26: end function
```

is initialized to the base method. If there is no replace callin the original function is scheduled otherwise the replacing role function.

The algorithm iterates over the active context instances either until all have been visited or a replace callin has been found. This is possible because of the total order (see Definition 8.4) that places callins in such a way that a replace callin always comes last. For each context instance c_i, the bindings are selected that are concerned with the base method in question. For each team, the bindings are accessed (lines 8–9). According to each binding the lifting and parameter mapping function *parmap* is selected and scheduled. By applying the parameter mapping function *parmap* to the base function the parameters of the base function are mapped to the respective parameters of the role function. The mapping of the base type to the role type is called *lifting*. The result is the lifted role function. Depending on the type of callin the lifted function is then sequentially composed to the respective group of callins. For example, when processing a before callin the lifted function is sequentially composed to the already processed before callins.

According to the OTJ Language Definition [HHM11, § 2.3.1.(b)] a base type $B \in \mathcal{B}$ can be *lifted* to a role type $R \in \mathcal{R}$ iff there is a team class $T \in \mathcal{C}$ with a binding $cb \in CB(T)$ s.t. $cb = (R, B, m_R, m_B)$. Lifting will be applied to base instances before each invocation of a role function. Furthermore, the compiler generates a parameter mapping function $parmap_{cb}$ that maps the signature of m_B to the signature of m_R in such a way that $sign(parmap_{cb}(m_B)) = sign(m_R)$. This is trivial if the signatures of m_B and m_R are equal. Otherwise, the parameter mapping must be provided by the user according to the OTJ Language Definition [HHM11, § 3.2].

Definition 8.5 (Lifting and Parameter Mapping)**.** Given two functions $m_B \in M(B)$ and $m_R \in M(R)$, a team $T \in \mathcal{C}$, and a callin binding $cb \in CB(T)$ with $cb = (R, B, m_B, m_R)$ that relates both functions. The function signatures are as follows. We have $sign(m_B, B) = T_{B_1} \times \cdots \times T_{B_n} \to T_{B_R}$ and $sign(m_R, R) = T_{R_1} \times \cdots \times T_{R_m} \to T_{R_R}$. Given the lifting function $liftTo_B^R : B \to R$ that maps base instances to their respective roles. The user-provided function $parmap_{cb}$ must be defined for each binding cb. It is defined as follows: $parmap_{cb} : (T_{B_1} \times \cdots \times T_{B_n} \to T_{B_R}) \to (T_{R_1} \times \cdots \times T_{R_m} \to T_{R_R})$ s.t. $\forall i, 1 \leq i \leq m$ there is a parameter mapping $\varphi_i^{cb} : T_{B_1} \times \cdots \times T_{B_n} \to T_{R_i}$.

It is not possible to schedule the whole dispatch plan including all role functions that must be executed when there is a replace callin to be scheduled. Before or after callins are sequentially composed to already scheduled before and after callins. Scheduling finishes when a replace callin has been processed because the callin might have a `base` call statement. As the statement is part of the body of the callin the role functions "deeper" in the stack can not be sequentially composed to the current schedule. For the base call (see `callNext` in Figure 7.2) a new dispatch plan will be created that has the index set to $i + 1$ when i callins had been scheduled prior.

8.3.2 Polymorphic Dispatch Plans

The schedule as discussed before must be recomputed for each reconfiguration where the activation or deactivation of a team instance changes the elements of active teams in γ and results in a new configuration γ' [SC19]. To reduce recomputations one could imagine to cache $(\gamma, \mathcal{P})$ and reuse the dispatch plan $\mathcal{P}$ whenever the runtime state again is γ [SC20]. However, this artificially limits the approach because this example defines the necessity to reuse a dispatch plan based on the instances of active teams in γ and not by their types. Let us consider the types of each of the contexts in the runtime state γ. Given $\gamma = (c)$, $\pi_0(\gamma) = c$, and $isa(c) = T$. Deactivating the team instance c results in a change from γ to the state $\gamma' = ()$. Subsequent activation of another team instance c' with $isa(c') = T$ results in a new state $\gamma'' = (c')$. One could say that γ and γ'' are *structural equivalent* as each type of both states are equal. Since we only compare the types of the runtime states it makes sense to define an environment that captures the types of the runtime state as defined in Definition 8.6. Instead of caching $(\gamma, \mathcal{P})$ it is sufficient to only cache the environment with the dispatch plan: $(\Gamma(\gamma), \mathcal{P})$.

Definition 8.6 (Environment)**.** Given a runtime state $\gamma = (c_0, \cdots, c_{n-1})$ we define the environment of γ as $\Gamma(\gamma) = (isa(c_0), \cdots, isa(c_{n-1}))$.

In class-based OOP we most often find receiver-type polymorphism where a polymorphic reference is bound to the runtime type of the receiver object. What we just discussed is the extension of the notion of polymorphism to a set of context instances that bind callins to a polymorphic call site. The reference is polymorphic concerning the environment $\Gamma(\gamma)$. We define this notion of polymorphism using the structural equality of runtime states in Definition 8.7. Polymorphism is decided by the equality of the types of contexts in the runtime state. The order of the contexts in γ (see Definition 8.4) is important because this decides the order of applicable bindings.

Definition 8.7 (Structural Equality of Runtime State)**.** Given two runtime states γ and γ' with n elements. We say γ is structurally equal to γ' iff. $\Gamma(\gamma) = \Gamma(\gamma')$. We write $\gamma \cong \gamma'$.

The definition of equality is still rigid because we compare whole runtime states. As we have seen in the previous section a dispatch plan can only evaluate the runtime state up to the first occurrence of a replace binding. Thus, all subsequent contexts will belong to another dispatch plan and are irrelevant to the former comparison.

To increase reuse one could relax the Definition 8.7 by only comparing subsets of the runtime states for structural equality. In simple words, one would compare each runtime state that contributed to a dispatch plan at each index with the contexts of another runtime state for type equality. If that holds the dispatch plan can be reused and we say that a runtime state γ is *partially* structural equivalent.

Definition 8.8 (Partial Structural Equality of Runtime State)**.** Given a dispatch plan $\mathcal{P}$ that resulted by starting evaluation of γ at index k and ended at $l = k + |\mathcal{P}|$. This is captured by $\gamma'' = \prod_l^k(\gamma)$. The new runtime state γ' has fewer contexts, s.t. $|\gamma'| < |\gamma|$ and $|\gamma'| = |\gamma''|$. γ' is partially structural equal to γ iff. $\Gamma(\gamma') = \Gamma(\gamma'')$ are structural equal. We write $\gamma' \lesssim \gamma$.

Theorem 8.9. For two dispatch plans $(\gamma, \mathcal{P})$ and $(\gamma', \mathcal{P}')$ where $\gamma' \lesssim \gamma$ it holds that $\mathcal{P} = \mathcal{P}'$. Without loss of generality, this also holds for $\gamma' \cong \gamma$.

We conclude that for two runtime states γ, γ' a dispatch plan can be reused if $\gamma' \lesssim \gamma$ because every context's type in γ' is equal to γ when starting comparison with γ' at an index k. Values of k are not chosen arbitrarily. k is bounded by either the start index $k = 0$ of the resolution of a callin (`callAllBindings` in Figure 7.2) or the start of the resolution of a base call (`callNext` in Figure 7.2) where $k = |\mathcal{P}|$ is the length of the preceding dispatch plan $\mathcal{P}$.[24]

If we apply Definition 8.8 entirely to environments we get that an environment is partially equal to another if the former is contained in the latter. This finally provides us with the extension of the notion of receiver-type polymorphism to the set of context types that allows late binding callins to a role-polymorphic call site. Thus, the reference is role-polymorphic concerning the environment $\Gamma(\gamma)$. As a result, we can formulate an extension of Polymorphic Inline Cache (PIC) to contextual roles which implementations may use as another optimization on top of optimizing contextual role dispatch with dispatch plans [SKC22].

8.4 Runtime Code Generation and Linking in a Java Virtual Machine

The JVM has multiple phases during the execution of a program where parts under execution may be (re)loaded. By generating code at "*runtime*" we mean the generation of code *after* classes have been loaded. With code generating we mean the generation

[24]If $\mathcal{P}$ already followed another dispatch plan $\mathcal{P}'$ then k is defined as the sum of the length of all previous dispatch plans.

of Bytecode or a similar representation of code that can be read and executed by a JVM. For example, there are ways of redefining the code of method bodies at runtime. The extension of a JVM to allow the redefinition of method signatures at runtime has been shown [WWS10]. Our approach requires the generation of code for each call site bound by a callin separately. Generating code and reloading the respective method bodies each time for each call site will incur an unacceptable overhead. This approach would also require to repeat this process every time a context changes. Using only features available in a standard JVM there is another facility to generate code at runtime. The facility allows the installation of runtime-generated code into a call site and can subsequently execute it as any other function implemented in Bytecode.

8.4.1 InvokeDynamic Bytecode Instruction

Instead of implementing super instructions for role invocations in the VM, we use the `invokedynamic` bytecode instruction [Ros09b] to link and invoke user-provided, run-time generated code. The bytecode was originally introduced to support dynamic languages on the JVM. The instruction is an invocation bytecode that behaves differently than the other invocation bytecodes such as `invokevirtual` for virtual function calls or `invokespecial` for static or private function invocations. During bytecode generation, a reference to a static method and the signature of that method is bound to the instruction–the *bootstrap* method.[25] To generate a valid `invokedynamic` bytecode instruction the types of the arguments applied to that function must adhere to the signature. Upon the first invocation, the call site is not set up, yet, and the static function is called. The bootstrapping process returns a `CallSite` which represents an object that may contain a reference to a method when invoked. The returned call site will be installed in the place of the `invokedynamic` instruction. Further invocations will use the contained reference and the dynamic arguments as defined in the `MethodType` provided to the bootstrap method. Dispatching the method invocation is subject to the user-defined implementation of the call site object.

8.4.2 Method Handles and Method Types

To describe how to resolve a method invocation to a call target the `OpenJDK` "*the DaVinci machine*" project [Ros09a] defines two central Java classes, namely `MethodHandle`

[25]The static method is required to at least provide the following arguments: `(Lookup, String MethodType)` where `Lookup` captures the visibility and lookup capabilities that are granted to the function resolution, the `String` holds a readable "*name*" of what is called, and the `MethodType` captures the argument and return types required by the returned call site. The bootstrap method must return a `java.lang.CallSite` object.

and `MethodType`.[26] A `MethodHandle` represents a typed, directly invocable reference to a method or field that may be subject to transformations to (individual) arguments or their return values. A `MethodHandle` can be constructed either via reflection or by using a `Lookup` object that provides lookup capabilities according to the privilege granted to the object. Given a class name, method name, and method type a reference to said method is returned. Given enough privilege, a lookup object may even look up and return method handles to private methods which is beyond what the Java language specification would allow.

A handle to a method could be directly executed given the arguments are available. The arguments of method handles may also be transformed by *method handle combinators*. Depending on the type of combinator, it takes as input a method handle, an integer denoting the position of the argument of the input method handle that is transformed, and a method handle to the transformation function itself. The result is a new method handle that may be executed or transformed again. The method handles may be viewed as the root of a small graph of combinators whose leaves are direct references to other bytecoded methods [Ros09b]. We give an overview of argument transformations provided by JSR 292 to build such a graph in Table 8.1. To implement variable arguments [27] the method handle combinator `asCollector` allows to define how many arguments starting from which position will be combined. Using `asSpreader` one can spread the elements from the array again as positional arguments. If a method does not need an argument one can use `dropArgument` to ignore unused arguments. With `filterArguments` one can compose a method handle with filter functions that are applied to the declared arguments. The combinator `foldArguments` allows to precompute a function on the arguments and adjoins the result into the given method handle's parameters. As we show later, we use this function to realize the sequential composition introduced in Algorithm 1. The argument adapters also bind values to arguments and reorder arguments which possibly drop and/or duplicate arguments.

A `MethodType` represents the arguments and return type accepted and returned by a method handle. Method types must be properly matched between a method handle and all its callers. The JVM enforces this matching among other things during the execution of `invokedynamic` instructions. Method types may be described in nominal form as a type descriptor. For example, the class `Account` can be either referenced by its reflective class `Account.class` or in its nominal form as the string "`LAccount;`".

[26]Especially the Java Specification Request (JSR) 292 introduces the features that may be used by dynamic programming languages that target the JVM.

[27]Variable arguments (also *varargs*) are denoted in Java as the last parameter's type with an additional three dots, e.g., `DataType... name`. This allows a method to accept zero to many arguments of the declared type which are collected into an array of `DataType`.

Table 8.1: Overview of `MethodHandle` argument adapters used in the implementation of Dispatch Graphs adapted from [Ros09b].

Combinator	Description of argument transformation
asCollector	collect N trailing arguments as array (introduce varargs)
asSpreader	spread N elements as positional arguments (eliminate varargs)
dropArgument	ignore consecutive N
filterArguments	unary compose $f(g(x_1), h(x_2), \cdots, x_n)$
foldArguments	adjoin $f(g(x_1, x_2, \cdots, x_n), x_1, x_2, \cdots, x_n)$
insertArguments	bind N values to arguments
permuteArguments	reorder (possibly drop and/or duplicate)

8.4.3 Runtime Code Generation with Method Handle Graphs

Method Handles, method types, and method handle combinators are ordinary Java objects and functions. However, optimizing method handle calls is possible because the method handle graph structure is immutable and scrutable; it can be walked by the compiler [Ros09b]. For example, the graph can be inlined whenever the root method handle can be constant-folded. This behavior is specific to the JVM-dependent implementation [TR10; XBH16]. We describe the behavior specific to our understanding of how the `OpenJDK`/`HotSpot` implements it. The `OpenJDK`/`HotSpot` implementation uses native [28] method implementations that bridge the gap between the host language Java and the internal representation of classes, objects, references, and values inside the JIT compiler and other VM modules. This enables the JVM to comprehend, generate code, and optimize the method handle graph. The resulting code is stored in the section of the JVM heap allocated to the code cache.

The `invokedynamic` bytecode is implemented internally via runtime bytecode generation. Given a method handle graph, the JVM traverses the graph and generates internal bytecodes before optimizing the code [TR10]. The bytecode is parsed and transformed into the internally used Static Single Assignment (SSA) form [BP99] and subsequently compiled by reusing the compiler infrastructure of the JIT compiler.

[28]Native means that there is a C/C++/ASM implementation that may be called via Java Native Interface (JNI). In this case, the implementation is inside the JVM `HotSpot` JIT compiler.

8.5 Dispatch Graphs

This section presents the implementation of dispatch plans and the extension of Polymorphic Inline Cache (PIC) to contextual roles for ObjectTeams/Java (OTJ) presented in Section 8.3. We begin by describing how code can be efficiently generated and linked at runtime in a JVM. Afterward, the implementation of dispatch plans is shown. Last the usage of guards to represent call sites polymorphic to a set of contexts.

8.5.1 Runtime Generated Role Dispatch Code

We reimplemented the complete dispatch logic of ObjectTeams/Java based on the `invokedynamic` bytecode. The proposed implementation uses the runtime code generation provided by `invokedynamic` to generate a graph from the resulting dispatch plan that can be optimized and re-executed by the JVM. Thus, we coined the prototypical implementation of JIT generated code from dispatch plans as *Dispatch Graphs*. The approach requires a mechanism to essentially guard and invalidate the compiled code upon the invalidation of assumptions. In Figure 8.1 we introduce the architecture as a UML class diagram. Orange-colored classes represent classes from the `invokedynamic` ecosystem such as `CallSite` from the package `java.lang.invoke` or `DynamicLinker` from the package `jdk.dynalink`. While `CallSite` is responsible for bridging the Java language ecosystem and JVM internals the `DynamicLinker` is responsible for realizing linking for language runtimes implemented as `GuardingDynamicLinker`. We implemented the ObjectTeams/Java runtime into this framework depicted with yellow-colored classes in Figure 8.1. This required us to provide a static bootstrapping method which we implemented in the class `CallinBootstrap` for `callAllBindings` and `callNext`.

In the *bootstrapping phase* the `invokedynamic` is (normally) executed for the first time. The process is shown in the UML sequence diagram in Figure 8.2. The framework from the package `jdk.dynalink` provides a dynamic linker that either forwards requests or *relinks* the call site when required. It holds all applicable concrete linker implementations to which it will delegate requests. The `CallinCallSite` holds a *target* method handle to which it delegates all further requests. The linker initializes the call site and installs a handle to the `relink` method. The result of the bootstrapping is a call site object from `java.lang.invoke` installed into the call site.

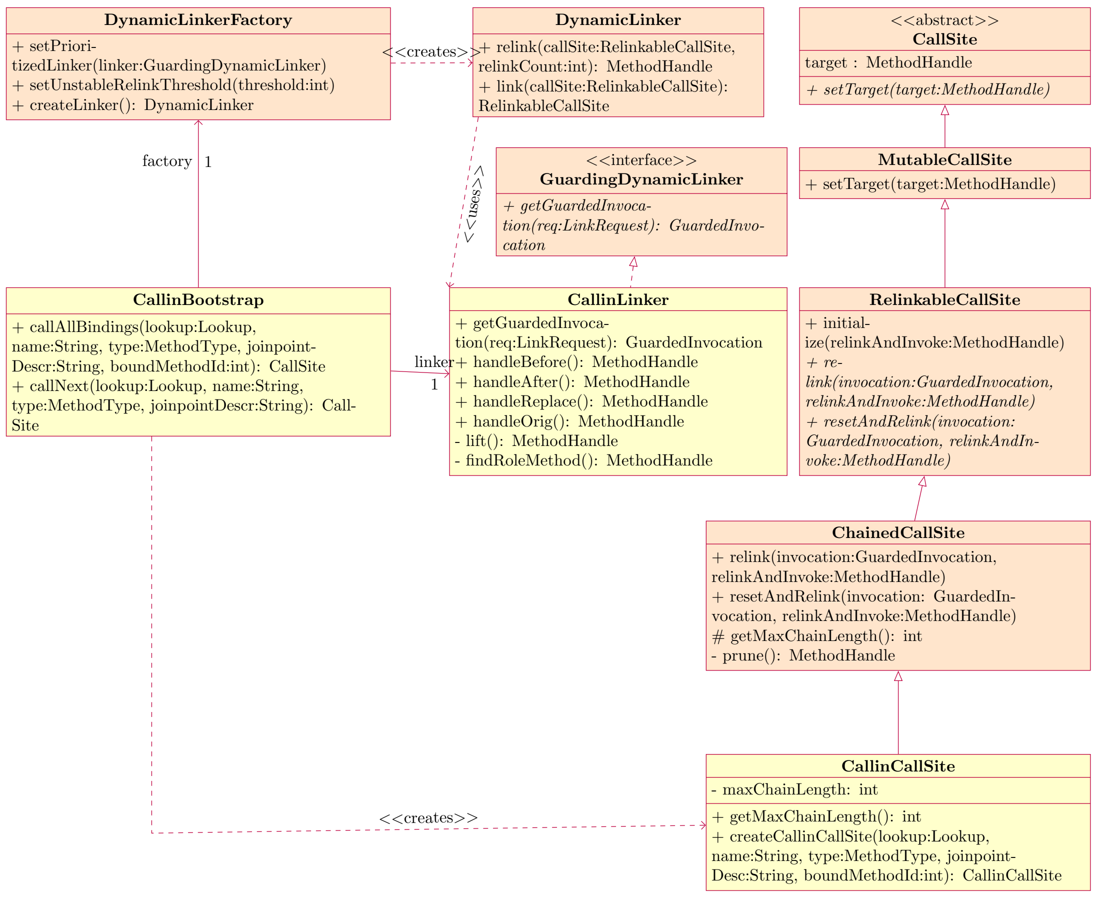

DynamicLinkerFactory
+ setPrioritizedLinker(linker:GuardingDynamicLinker)
+ setUnstableRelinkThreshold(threshold:int)
+ createLinker(): DynamicLinker
<<creates>>
DynamicLinker
+ relink(callSite:RelinkableCallSite, relinkCount:int): MethodHandle
+ link(callSite:RelinkableCallSite): RelinkableCallSite
<<abstract>>
CallSite
target : MethodHandle
+ setTarget(target:MethodHandle)
factory
1
<<uses>>
<<interface>>
GuardingDynamicLinker
+ getGuardedInvocation(req:LinkRequest): GuardedInvocation
MutableCallSite
+ setTarget(target:MethodHandle)
CallinBootstrap
+ callAllBindings(lookup:Lookup, name:String, type:MethodType, joinpointDescr:String, boundMethodId:int): CallSite
+ callNext(lookup:Lookup, name:String, type:MethodType, joinpointDescr:String): CallSite
linker
1
CallinLinker
+ getGuardedInvocation(req:LinkRequest): GuardedInvocation
+ handleBefore(): MethodHandle
+ handleAfter(): MethodHandle
+ handleReplace(): MethodHandle
+ handleOrig(): MethodHandle
- lift(): MethodHandle
- findRoleMethod(): MethodHandle
RelinkableCallSite
+ initialize(relinkAndInvoke:MethodHandle)
+ relink(invocation:GuardedInvocation, relinkAndInvoke:MethodHandle)
+ resetAndRelink(invocation: GuardedInvocation, relinkAndInvoke:MethodHandle)
ChainedCallSite
+ relink(invocation:GuardedInvocation, relinkAndInvoke:MethodHandle)
+ resetAndRelink(invocation: GuardedInvocation, relinkAndInvoke:MethodHandle)
# getMaxChainLength(): int
- prune(): MethodHandle
<<creates>>
CallinCallSite
- maxChainLength: int
+ getMaxChainLength(): int
+ createCallinCallSite(lookup:Lookup, name:String, type:MethodType, joinpointDesc:String, boundMethodId:int): CallinCallSite

Figure 8.1: A UML class diagram providing a top-level view of how the architecture of dispatch graphs (yellow) is embedded in JVM-provided classes of the `invokedynamic` ecosystem (orange).

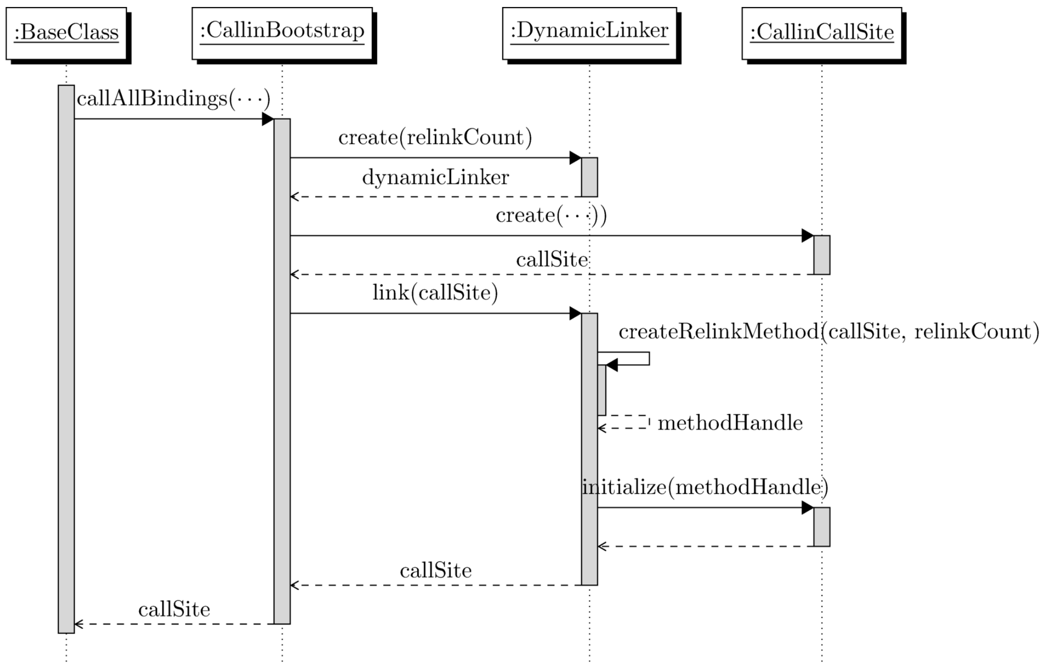


Figure 8.2: A UML sequence diagram providing an overview of the bootstrapping of a base method with roles. We omitted some details for brevity.

8.5.2 Role Method Invocations as Revokable and Dynamic Call Site

Any change in the set of active teams impacts the call sites that have to apply role dispatch. To deal with changing call targets, the approach requires a dynamic call site that can invalidate linked code and relink *new* code upon the next invocation. We, therefore, extended the `RelinkableCallSite` from `jdk.dynalink` which already provided a mechanism for *relinking*. Its subclass `ChainedCallSite` implements a polymorphic inline caching strategy. Upon linking it builds a chain of cascading method handles where a handle fails from one to the next. The implementation of polymorphic dispatch plans is discussed in Section 8.5.4. If too many handles are chained (defined by `maxChainLength` in `CallinCallSite`) the oldest entry will be *pruned*.

The UML sequence diagram in Figure 8.4 shows the relinking process. The initialized call site objects hold a method handle pointing to the `relink` method of the dynamic linker. The `invokedynamic` instruction will execute the installed method. Role methods are linked by the `CallinLinker` according to the joinpoint. It returns a `GuardedInvocation` that encapsulates a method handle pointing to the root of the

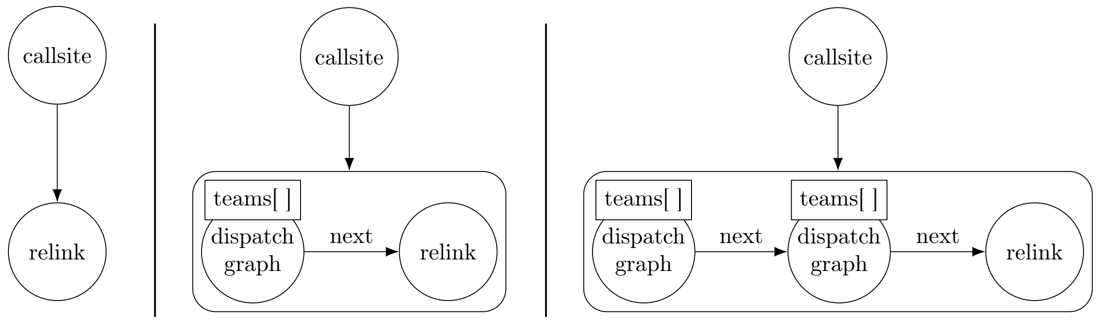


Figure 8.3: Run-time changes at the call site of a bound method.

dispatch graph. The root of the dispatch graph is the entry point to the execution at the call site with the arguments present on the call stack. The JVM can (abstractly) interpret the Intermediate Representation (IR) to yield whether the provided and required signature of the call site is fulfilled by the graph. Last the linker checks the call site for stability. A call site is stable when the threshold for relinks is not reached, yet. Depending on the call site-local count of relinks the call site is either relinked effectively appending the invocation or—if deemed *unstable*—*reset*. A reset throws away all linkages of the call site. The *next* invocation flags the call site as unstable and linkers may react in the linking process.

In our implementation, when the flag is set, the `CallinCallSite` will *degrade* and link the original linking functions from OTJ which is either `callAllBindings` or `callNext` [SKC22]. This is possible because dispatch graphs are a backward-compatible replacement of the original dispatch implementation in OTJ. The `invokedynamic` call site accepts the same signature as the original implementation.

8.5.3 Linking On-Demand Compositions of Role Methods

The process of creating a dispatch plan has been discussed in Section 8.1. This section discusses points of the method lookup and graph creation process to realize Algorithm 1 and aspects of the implementation. We present the dispatch graph as a high-level IR generated from a dispatch plan. The IR is inspired by the sea-of-nodes notation [CC95] and is subsequently optimized by the JVM. Each white box represents a function, while red boxes represent special nodes such as the beginning of a basic block or return instructions. Turquoise ellipses represent data. The red edge defines the control flow while the dashed edges define the data flow. The annotation on dashed edges defines the order of incoming arguments. A partial order among nodes connected by the

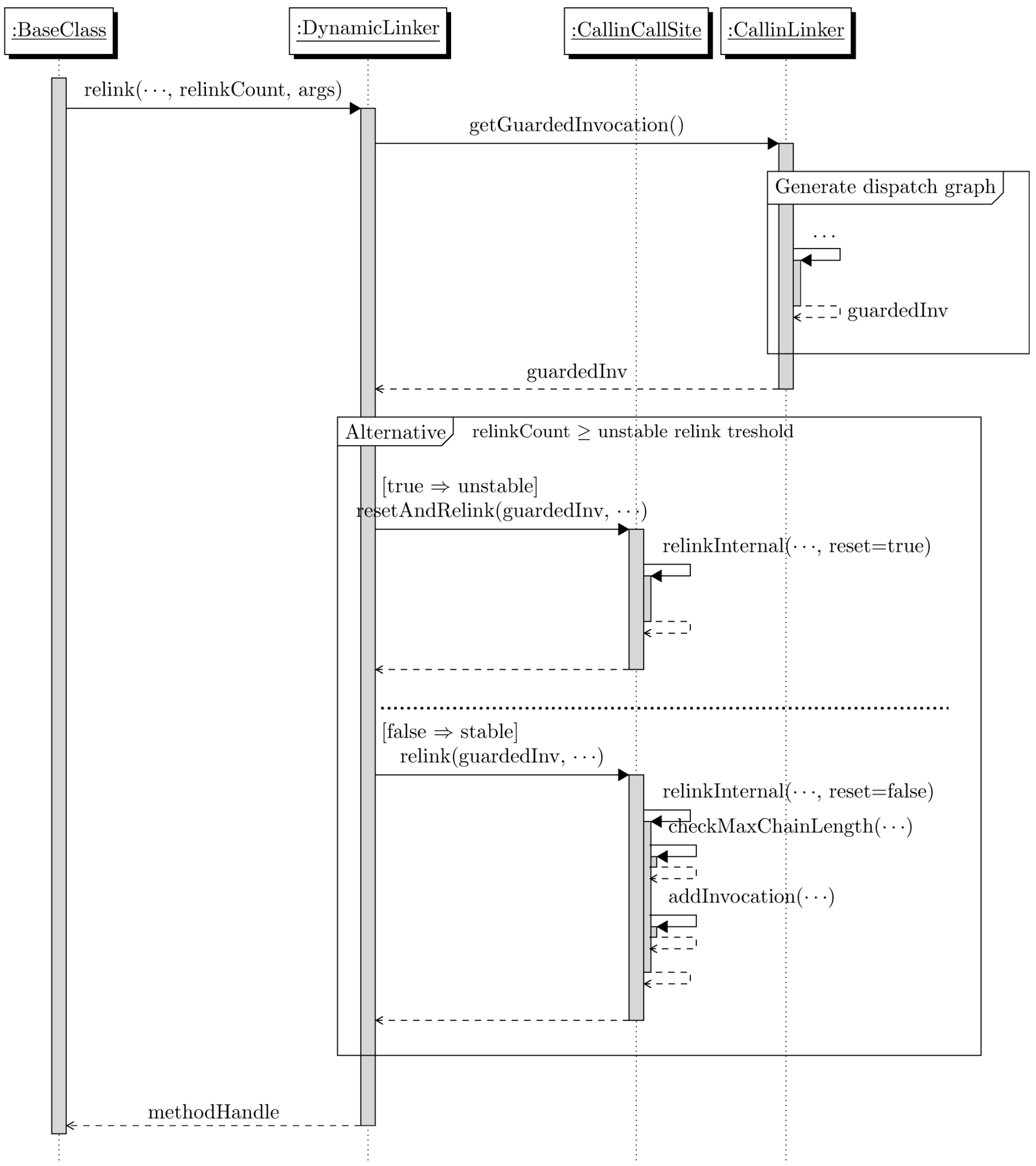

:BaseClass
:DynamicLinker
:CallinCallSite
:CallinLinker
relink(· · ·, relinkCount, args)
getGuardedInvocation()
Generate dispatch graph
. . .
guardedInv
guardedInv
Alternative
relinkCount ≥ unstable relink treshold
[true ⇒ unstable]
resetAndRelink(guardedInv, · · ·)
relinkInternal(· · ·, reset=true)
[false ⇒ stable]
relink(guardedInv, · · ·)
relinkInternal(· · ·, reset=false)
checkMaxChainLength(· · ·)
addInvocation(· · ·)
methodHandle

Figure 8.4: A UML sequence diagram providing an overview of the relinking process of a base method with roles immediately after initialization. We omitted some details for brevity.

$$\begin{array}{ll}
 & (\text{Base}, \boxed{\text{Team[ ], int,}}\ \text{int[ ], Object[ ]})\ \text{void} \triangleright \textit{apply}\ \texttt{teamGetter:(Team[ ], int)Team} \\
\rightarrow & (\text{Base, Team}, \boxed{\text{Team[ ], int, int[ ],}}\ \text{Object[ ]})\ \text{void} \triangleright \textit{dropArguments} \\
\rightarrow & (\text{Base, Team}, \boxed{\text{Object[ ]}})\ \text{void} \triangleright \textit{unpack arguments} \\
\rightarrow & (\boxed{\text{Base, Team,}}\ \text{float})\ \text{void} \triangleright \textit{permute arguments} \\
\rightarrow & (\boxed{\text{Team, Base,}}\ \text{float})\ \text{void} \triangleright \textit{apply}\ \texttt{lift:(Team, Base)Role} \\
\rightarrow & (\text{Role, float})\ \text{void}
\end{array}$$

Figure 8.5: A sequence of derivations (represented as $\rightarrow$) applied to resolve a base method signature to the respective callin signature. Arguments that are rewritten are highlighted with a $\boxed{\text{box}}$, $\triangleright$ names the applied rewrite rule. We show the resulting signature (method type) after applying the rewrite.

control flow edges defines a possible execution order, one of which is highlighted by the annotation on the vertices.

The implementation can access the MOP of OTJ [29] by accessing the arguments on the call stack during resolution. The call stack comprises the instance of the base class whose bound function was called, an array of team instances that are currently active for the given call site, an array of callin identifiers representing actual callins, and the current index. The index presents the current depth and is used to access both arrays. As long as the linking proceeds we increment the index and use the values from the call stack to collect the binding information of a given callin from the `TeamManager`.

Next, the callins must be discovered using the information from the `Binding`. We discuss before and after callins together as they both behave similarly and look at replace callins separately. Each binding that is resolved will be attached to its respective subgraph which will be composed into a single graph at the end. The discussion is focused on the graph creation process.

8.5.3.1 Before and After Callins

The before and after bindings share the resolution process but will be inserted into different parts of the graph. In Figure 8.5 the effects of the transformations on the signatures (method types) are shown. First, a method handle pointing to the role method is created by looking up the callin method using its name and signature stored in the binding. The method handle only points to the method but is not bound to a

[29] We introduced the Metaobject Protocol of ObjectTeams/Java in Chapter 7.2. An overview is given in Figure 7.1.

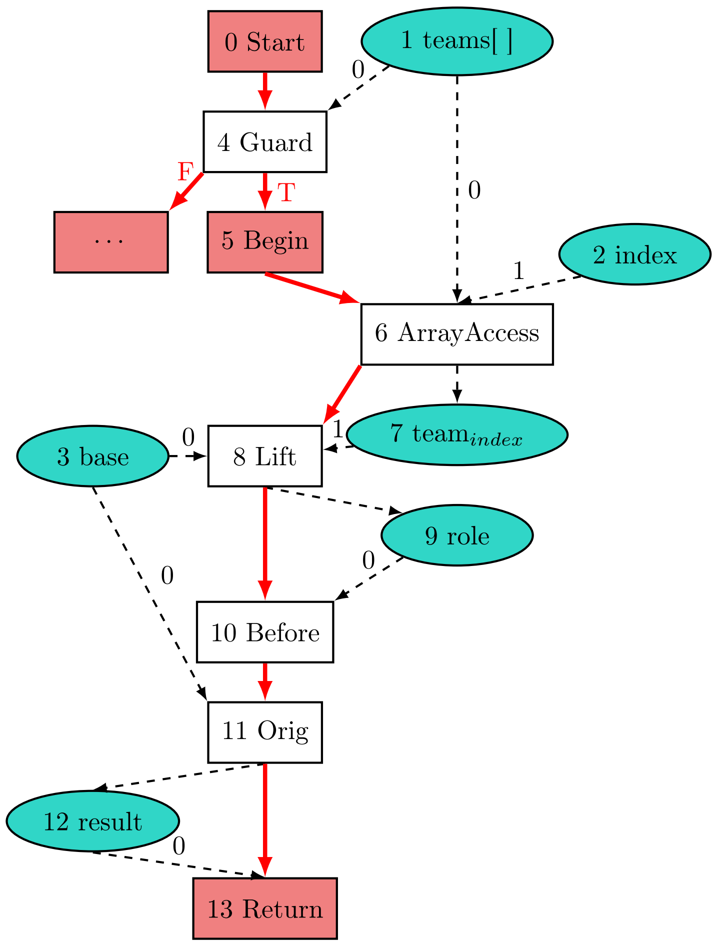


Figure 8.6: A IR representation of a dispatch graph composing before callins with the base function (adapted from [SKC22]).

Listing 8.20 A sketch of the implementation of the sequential composition of after callins using method handle combinators.

```
MethodHandle afterComposition = null;
...
case AFTER:
  MethodHandle handleAfter = computeHandleAfterCallin(...);
  afterComposition =
    MethodHandles.foldArguments(afterComposition, handleAfter);
  break;
```

concrete instance, yet. The computation of that instance is part of the graph. Second, a method handle to the lifting function for that particular base-role relationship is resolved. Since the lifting function is compiled into the enclosing team class it must be called on the respective team instance the role belongs to. To achieve this we access the array of teams from the call stack and get the team instance at the current index. The lifting function uses that respective team instance when being executed. The next argument of the lifting function is the base instance that has to be lifted which resides on the call stack. During invocation, it will be consumed and replaced by the respective role instance. The graph representing the procedure is shown in Figure 8.6. The resulting method handle will be sequentially composed with already resolved callins of the respective type. A sketch of the sequential composition is given in Listing 8.20.

8.5.3.2 Replace Callins

In contrast to before and after callins—which can only have `void` specified as return type—replace callins can specify arbitrary return types. This requires us to treat them differently when composing the complete dispatch graph. If there exist after callins then the result of the replace callin (or original base function) must be *tunneled* past the after callins to be returned to the callee. After callins do not produce any results and may not change the control flow.

A sketch of the implementation including result tunneling is given in Listing 8.21. If there exists a replace callin `result` points to the replace callin, otherwise to the base function. We guide through the rest of the code in the figure in reverse. First, the function `result` points to is executed, and the value computed is added to `afterComposition`. The method handle `afterComposition` then executes the composition of after callins. Ultimately, the result value will be returned by the call site due to the identity function.

Listing 8.21 A sketch of the implementation of result tunneling using method handle combinators.

```
MethodHandle result = (replace == null) ?
  handleOrig(desc, baseClass) : replace;
if (afterComposition != null) {
  MethodHandle returnWrapper = MethodHandles.identity(Object.class);
  MethodHandle returnWrapperDropped =
    MethodHandles.dropArguments(returnWrapper, 1,
      afterComposition.type().parameterList());
  afterComposition = MethodHandles.foldArguments(
    returnWrapperDropped, 1, afterComposition);
  result = MethodHandles.foldArguments(afterComposition, result);
}
```

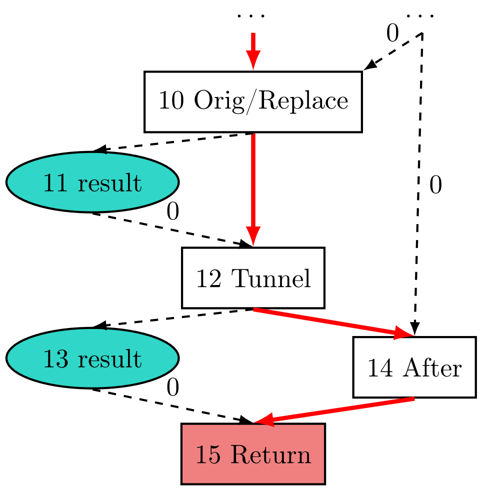


Figure 8.7: A IR representation of a dispatch graph presenting result tunneling (adapted from [SKC22]).

Listing 8.22 Implementation of the guard checking the Partial Structural Equality of Runtime State according to Definition 8.8.

```java
private static boolean testTeamComposition(final Class<ITeam>[] guardedStack,
  final ITeam[] runtimeStack, final int index) {
  if (runtimeStack == null ||
    index + guardedStack.length > runtimeStack.length) {
    return false;
  }
  for (int i = 0; i < guardedStack.length; i++) {
    int j = i + index;
    if (!guardedStack[i].isAssignableFrom(runtimeStack[j].getClass())) {
      return false;
    }
  }
  if(guardedStack.length == 0) {
    return runtimeStack.length == index + 1;
  }
  return true;
}
```

The graph resulting from Listing 8.21 is shown in Figure 8.7.

8.5.4 Polymorphic Inline Caches for Contextual Role Dispatches

Until now we ignored that a dispatch graph is only valid if the active teams remain the same. In Figure 8.3 the box with the label `teams[ ]` placed on top of a dispatch graph implies that a dispatch graph captures the runtime state it was created from. For each dispatch graph, a guard is created that will check the Partial Structural Equality of Runtime State according to Definition 8.8. The *guard* ensures that invalid lookup results will not be executed. Upon success the method handle pointing to the root of the dispatch graph is executed, otherwise, the next entry is checked. The guard itself is a method handle to a function that compares two runtime states for partial equality. The implementation is shown in Listing 8.22. The runtime state captured in `guardedStack` is already bound when the guard is created. In the beginning, we must check whether there are any active teams or if the guarded stack overflows the runtime stack. Then we check whether the teams of the guarded stack match with the teams from the runtime stack taking the current index into account. If there is any difference we report `false`. Last, we check whether we have guarded a call to the original base function.

8.6 Performance Evaluation

This section will evaluate the effectiveness of the implementation of Dispatch Graphs. To showcase the effectiveness of the optimizations we will compare different configurations.

8.6.1 Experimental Design

We shortly describe the design used for the experiments. Most of the setup is already described in Chapter 6. To coordinate the execution of the benchmarks we used the benchmark suite as presented in Section 6.2. Regarding the methodology, if not said otherwise, the same approach is followed as presented in Section 6.2.2.

The experiments were conducted on a Linux server with Ubuntu 20.04, 32GB RAM, and Core i7-9700T CPU. For execution, we used Oracle JDK 14.02 and GraalVM 20.2 with a heap size of 8GB heap. The different versions of ObjectTeams/Java are referenced by their year of publication, i.e., Classic 2019, to express ObjectTeams/Java published in 2019. The benchmarks in Section 8.6.2, Section 8.6.3, and Section 8.6.5 were implemented with Java Microbenchmark Harness (JMH) and measured for 10 iterations after 10 warmup iterations.

8.6.2 Instrumentation Overhead

Often conditional interception requires that residuals have to be evaluated contributing to the overall execution time even if there is nothing to adapt [Hau+05]. As we described in Chapter 7, the compiler generates role dispatch logic for each role that declares a binding. In this benchmark, we measured the overhead incurred by the instrumentation compared to the reference implementation ObjectTeams/Java. First, we measure the time to call a single method that has a registered binding but no active context instance. After evaluating that there is no active context instance the original function is executed. Second, a single context instance is activated and we measure the time to call the single method with the activated binding.

Figure 8.8 shows the geomean execution time. In the case of no operation (Figure 8.8 left) our approach is 2.5× faster than the reference implementation ObjectTeams/Java. The guard captures that there is no active context and the call site directly links to the

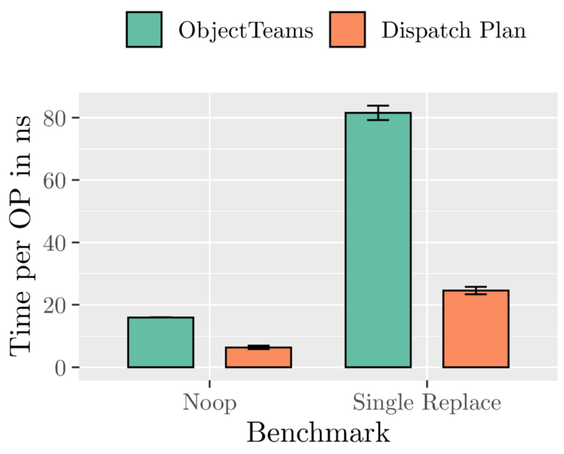


Figure 8.8: Comparison of no active contexts (noop) and a single context with a replace role function. Figure adapted from [SKC22].

original method. In the case of a single replace callin (Figure 8.8 right) our approach is 3.3× faster. The guard captures the active context, generates the dispatch graph accordingly, and links the call site directly to the replace callin. The same applies to the base call which directly calls the original method. The reference implementation always uses the stub methods to dispatch which explains the overhead.

8.6.3 Characteristics of Role Method Types

This benchmark evaluates the impact of different types of callins on the overall run time. We evaluate the characteristics of each variant of callin in isolation, i.e., use a synthetic benchmark only consisting of before, replace, and after. In these benchmarks, a context only provides a single type of adaptation that is subject to evaluation. We explore the impact of varying the amount of active context instances between 1, 10, and 100.

The results are shown in Figure 8.9. Due to the fixed dispatch scheme, the execution time of the original implementation in every scenario is almost the same. Our approach composes a different plan depending on the types of behavioral adaptations. Dispatch Plans are on average 2.9× faster (max 3.5×) in the case in which all bindings are of type before. Role methods of type after are executed on average 3.3× faster (max 3.7×) than the original implementation. While not changing contexts the guards could be

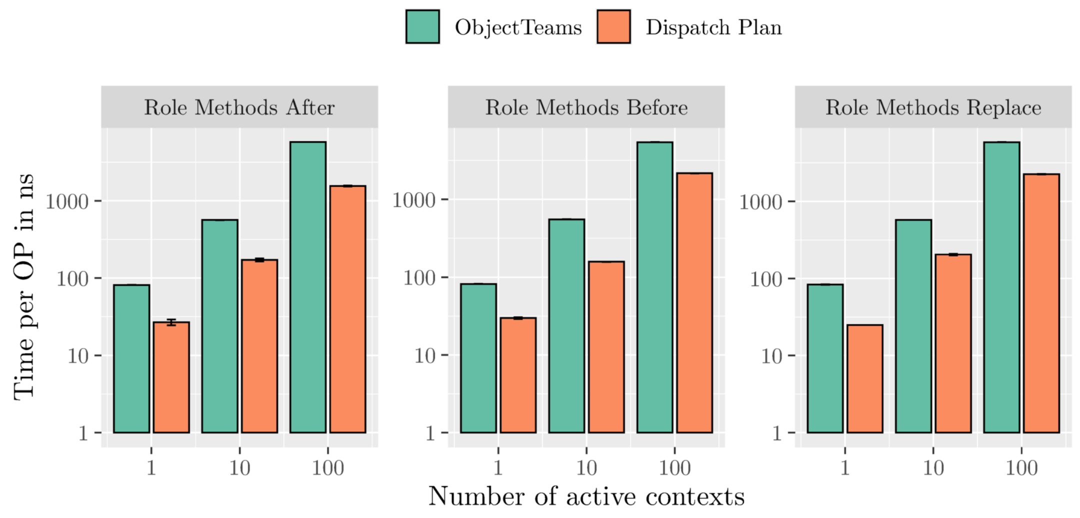


Figure 8.9: The contribution of different role function types (before, after, replace) to the overall execution time. Each type is measured with 1, 10, and 100 active contexts. Figure adapted from [SKC22].

completely reused allowing to execute replace callins on average 2.9× faster (max 3.4×).

For polymorphic dispatch plans without degradation speedups between 3.8× to 4.5× have been reported for static cases [SC20]. This is comparable to our results as the guards introduce more computations and jumps in the resulting code.

8.6.4 Impact of Polymorphism and Support for Partial Equivalence

In this Section, we compare the runtime performance of dispatch graphs in a static scenario and a dynamic scenario with the reference implementation of OTJ from 2019 and 2020. Please note, that for the Polymorphic Dispatch Plans and the reference implementation of OTJ (named `Classic 2020` in Figure 8.10), the Oracle JDK 14.0.2 was used, while the other implementations were run on the Oracle JDK 9.0.4. For this benchmark, the JVM was set to a maximum heap space of 4GB and to only use the server compiler. For comparison, an older version of the reference implementation of OT (`Classic 2019`) as well as `Dispatch Plans` [SC19] were also measured.

The benchmark used different problem sizes to evaluate the approaches on different inputs. In each benchmark, N persons have $2 \cdot N$ accounts (a CheckingAccount and a SavingsAccount). We are interested in the geometric mean execution time, normalized

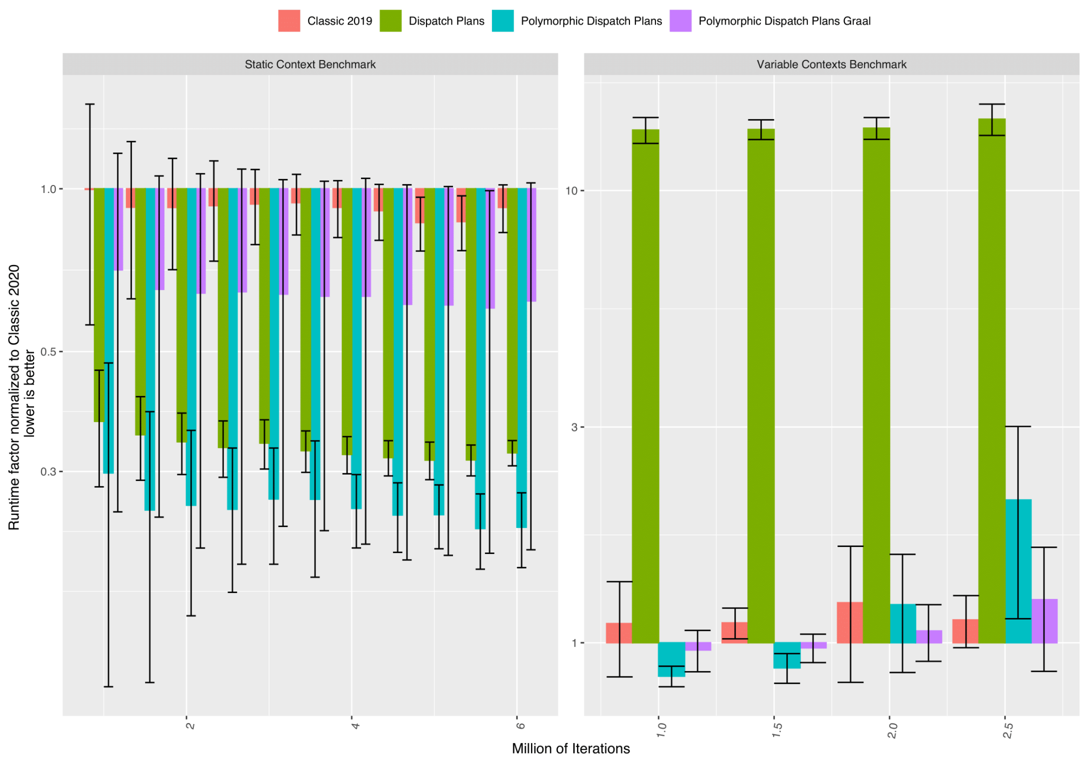


Figure 8.10: The bar chart shows the geometric mean of the run-time ratio at logarithmic scale, normalized to the classic implementation of Object Teams for the respective suites. The error bar shows the standard deviation of the run-time ratio. Figure adapted from [SC20].

to the reference implementation of OTJ, `Classic 2020`. To observe whether there are scalability problems, the problem sizes varied in the benchmarks. For example, the static case uses fewer role objects and could be scaled up to 6 million transactions. In the dynamic setting, the total number of transactions varied from 1.0 to 2.5 million.

The chart depicted in Figure 8.10 shows the results of the reference implementation of Object Teams (`Classic 2020`) compared to our proposed `Polymorphic Dispatch Plans`. The plot reports the geometric mean of the run-time ratio, i.e., the run-time factor. To compare to the current state of the art, the values are normalized to `Classic 2020`. The y-axis uses a logarithmic scale to highlight where each approach introduces a run-time overhead or improves over the current implementation. Lower values present better results. The error bar shows the standard deviation of the runtime ratio.

Figure 8.10 left shows how the static case benefits both approaches, dispatch plans as well as polymorphic dispatch plans. While dispatch plans compile the active teams directly into the DAG, polymorphic dispatch plans still require them to be present on the argument stack [SC20]. Because the call site is stable, the retrieval of this run-time data can be optimized. The optimization applied in [SC20] is to enforce copy-on-write optimizations for the data structure representing the active team stack. Polymorphic dispatch plans achieve a geometric mean speedup of 3.8× (max. 4.5×) compared to `Classic 2020`. They also improve over dispatch plans by 1.4×.

The execution of Polymorphic Dispatch Plans on the GraalVM 20.2 results in a lower performance. It improves over `Classic 2020` by 2.8× but is 2.6× slower than executed on JDK 14. We want the interesting reader also to note the high standard derivations across the different benchmarks when running on GraalVM. We assume that GraalVM does not, in the current implementation, optimize `invokedynamic` or graphs of method handles properly enough resulting in lower performance.

Figure 8.10 right shows the results of the dynamic setting. In this setting, we measured a 14× run-time overhead for Dispatch Plans compared to `Classic 2020`. Dispatch plans realize the reaction to changes in contexts by invalidating the generated graph triggering deoptimization mechanisms in the JVM. To reuse guarded dispatch plans, they require the stack of team instances not to change Polymorphic Dispatch Plans on the other side are designed to reuse generated dispatch plans. Due to the supported polymorphism, they can leverage the fact that the run-time stack of teams is *structurally* equivalent. For smaller problem sizes, the approach of role-polymorphic inline caches achieves a geometric mean speedup of 1.1×. For bigger problem sizes, the garbage collection starts to impact the overall execution time introducing more variance resulting in a *median* slowdown of up to 2.3× and a standard deviation of 0.95.

Executing Polymorphic Dispatch Plans on the GraalVM 20.2, which is built on OpenJDK 11, reduced the slowdown to up to 1.2×. To underpin the assumption that the variance is introduced by the garbage collector we increased the maximum heap size up to 8GB. As a result, the values level off at a geometric mean speedup of 1.1×. This result also corresponds to the observations when manually recording the running JVM in separate runs.

8.6.5 Instability and Graceful Degradation

When variability is high there is no benefit in operating a cache for dispatch plans. Cache entries would be evicted immediately resulting in high cache contention. This considerably impacts the achieved performance. To overcome these problems we provide a graceful degradation strategy to revert to the original dispatch whenever a call site becomes *unstable*. To evaluate the effectiveness of the degradation approach we designed a synthetic benchmark that permutates the set of active contexts. Each context contributes a before, replace, and after callin. Each time the same call site is called, a new dispatch plan is created capturing the new runtime state. We report the overall execution time across these permutations in Figure 8.11. We compare the execution time of the reference implementation OTJ 2020 against polymorphic dispatch plans with and without graceful degradation enabled.

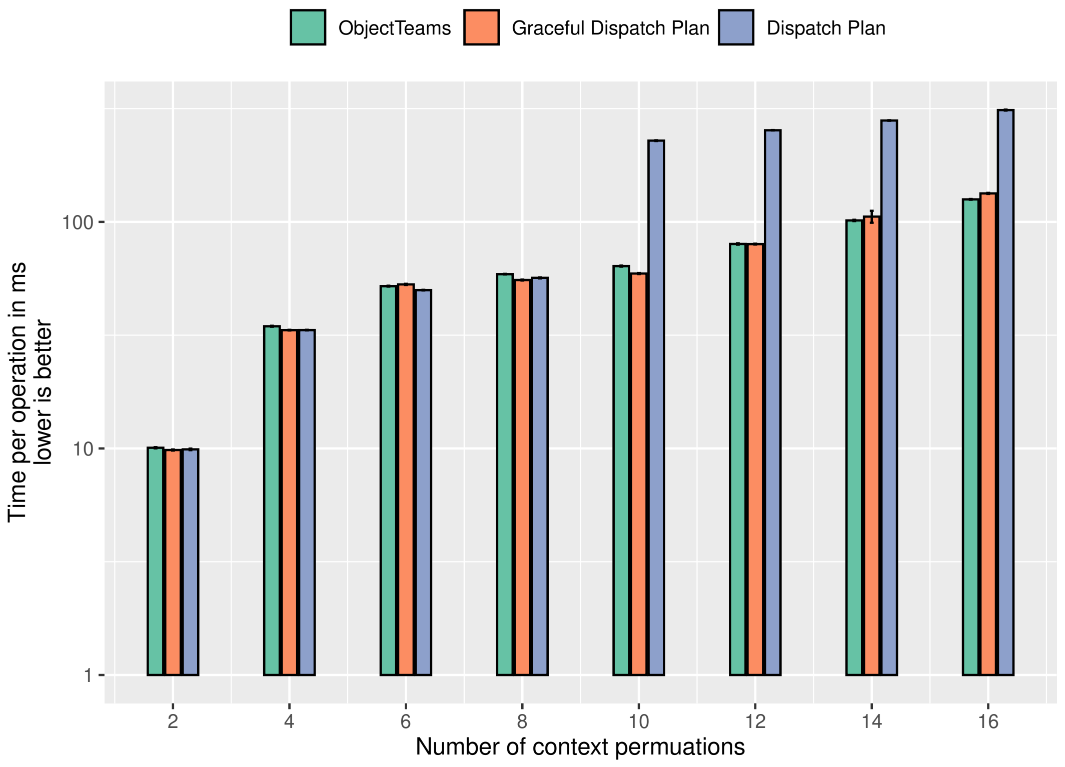


Figure 8.11: The impact of different permutations in the runtime state. The original approach is compared to dispatch plans and dispatch plans without graceful degradation.

Compared to the original implementation, our approach has on average the same execution time as the original approach since there is no possible reuse. However, without graceful degradation the execution time increases by up to 3.8×. This means that too much variability can be effectively countered by degrading to the original dispatch.

8.7 Conclusion

This chapter formalized the object model of ObjectTeams (OT) defining roles at run-time and introduced the Metaobject Protocol (MOP) used in the reference implementation to realize the OT model in the object-oriented programming language Java. We discussed how Partial Evaluation (PE) can resolve dispatches given instances of the reference object model at run-time. The result is a dispatch plan that captures the resolved role methods. A dispatch plan is only valid when the same types of contexts that contributed roles during resolution are still active. Such a constraint can be checked by a guard extending the notion of Polymorphic Inline Cache (PIC) to the runtime state of contextual role-oriented programs. Chaining guards enable the storage of multiple results in a call site. A call site can react to dynamic environments and gracefully degrade from a dynamic call site to a static call site negating the overhead from repeated evaluation.

The chapter also proposed an implementation for dispatch plans based on the `invokedynamic` bytecode and method handles. Method handles, like pointers to functions, can be used to build custom call graphs. The JVM provides special implementations for the components used to compose the dispatch graph. Thus, by describing role dispatch in an understandable format the JIT compiler can apply optimizations not possible when dispatch is described in the application logic. The limit of such a description is that it is not possible to actively guide the JIT compiler to apply optimizations or to declare custom optimizations that are to be applied.

9 A Virtual Machine Architecture for Contextual Roles

"Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less."
— Marie Curie

9.1 High-Performance Dynamic Language Runtimes

This section introduces the Graal JIT compiler as a representative of the class of method-based Just-In-Time compilers. We then introduce the Truffle DSL and compiler for high-performance dynamic language runtimes. Last we introduce Espresso, a meta-circular Java Bytecode Interpreter implemented as a Truffle DSL.

9.1.1 Just-in-Time Compilation

The success of managed programming languages with automatic memory management such as the Java programming language, to some degree, resulted from the guarantees to the security and portability of programs written in those languages. At that time, programs often were not transportable between architectures and required high efforts to maintain different implementations for each architecture. To deliver portability the proposed language runtimes are based on a *virtual instruction set*—an IR that programming languages are compiled to. For example, the Java programming language is compiled to Java Bytecode, which is an instruction set of a statically typed stack-based VM. First approaches were simply interpreting the IR at runtime leading to poor performance compared to compiled languages.

To deliver peak performance plus portability and security the VMs provided a new compilation model that turns the virtual instructions into machine code at runtime [Cra+97]. Nowadays, Just-In-Time compilation is a generic term representing a class of different approaches that compile a program (or specification) at runtime. For example, a runtime may rely on a complex compiler framework such as LLVM to compile and link code at runtime. We do not consider this kind of runtime compilation but connect the term Just-in-Time with one of the soft requirements, that is to achieve

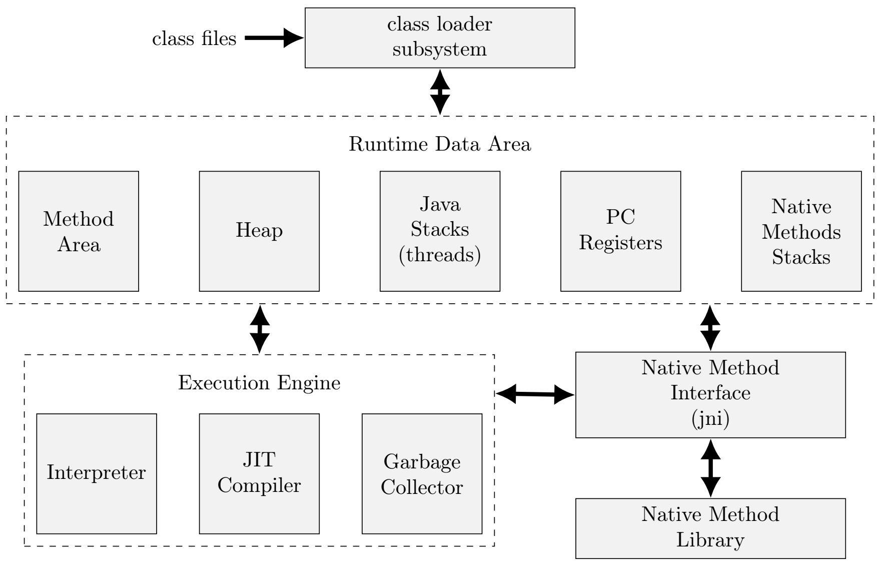


Figure 9.1: High-level Java Virtual Machine Architecture and Subsystems

compilation in a short period of time to make the impact of runtime compilation to the overall execution time as small as possible. Most often the JIT compilers employ faster algorithms to construct machine code trading compilation speed for resulting code efficiency. A great example is that JIT compilers often do register allocation using linear scan [PS99; WF10] instead graph coloring [MMI72] which in general should result in poorer execution performance but only requires asymptotical time-complexity of $\mathcal{O}(n)$ instead $\mathcal{O}(n^2)$.

To outweigh the compilation overhead of JIT compilation, only important chunks of the program are compiled. Thus, VMs provide interpreters and compilers that can cooperatively execute programs. Decision making whether a compilation may happen is based on heuristics that range from the size of the compilation unit, call counters reflecting the "*hotness*", or the depth of a method in the call stack. There exist different approaches to JIT compilation [MD15], where the unit of compilation ranges from method-based to trace-based. The Graal [Dub+13b] JIT compiler uses a method-based compilation model where the decision making for optimizations depends on heuristics such as method execution counts. Approaches such as PyPy [Bol+09], however, focus on compiling execution traces of the interpreter (meta-tracing) and evaluating the program under execution. The work proposed in this thesis builds on approaches using the former model of compilation. Thus, we will explain the compilation flow with the

Graal [Dub+13b] JIT compiler in mind.

An overview of the JVM architecture and some of its subsystems is presented in Figure 9.1. Class files—a binary format resulting from compiling Java code—contain bytecodes and meta-data. The *class loader subsystem* is responsible to load, check, and resolve the class files and to provide the respective bytecode to the execution engine via the *runtime data area* That subsystem consists of the *method area* where compiled code is stored, the *heap* which is used to store runtime objects and is managed by the *garbage collector*. The runtime area also comprises the stacks of running application threads, as well as registers that store various program counters. Last, it also stores the stacks of native methods which have been called via the Java Native Interface (JNI).

For our work we care most for the *execution engine* which is comprised of the interpreter, possibly multiple JIT compilers with different optimization goals (startup vs peak performance), and various kinds of garbage collectors. The JVM is a stack-based virtual machine and the interpreter manipulates the stack while executing the bytecodes. When a method is selected for compilation the Java Bytecode is transformed into a graph-based IR [Dub+13a; Dub+13b] inspired by Sea-of-Nodes (SoN) representation [CP95] and subsequently compiled by the JIT compiler. JIT compilers can *speculatively optimize* units under compilation based on collected statistics of the running application. This reduces the nodes in the IR to be compiled, for example, by only compiling likely branches or by specializing methods and method calls towards observed types. It may also allow more aggressive optimizations that only are possible due to collected runtime information [SWM14]. This optimizations require a deoptimization mechanism to be present that returns from compiled to interpreted code starting the information gathering anew [Flü+17; DWM14]. A particularity of Graal is that it is written in Java itself instead of C++.[30] The idea of a meta-circular VM is not new and already has been explored with the Jikes RVM [Alp+05] and Maxine VM [Wim+13]. Each approach provides its own mechanisms to bootstrap the Java-based compiler. However, recent advances in VM optimization research have opened the door for these kinds of VMs to be applied in commercial products. At the time of writing this thesis meta-circular VMs reach peak-performance comparable to state-of-the-art implementations but require more time of *warm up*, i.e., to optimize the important parts of a program.

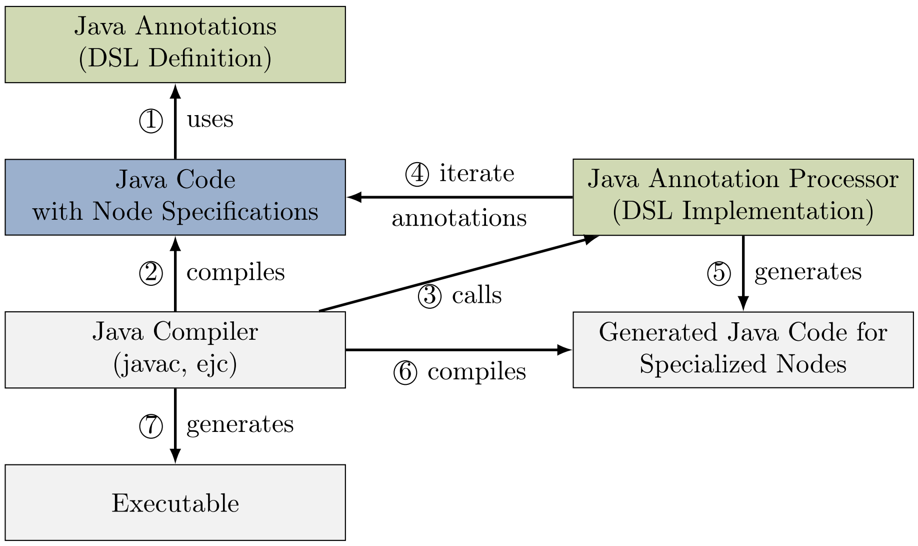

Java Annotations (DSL Definition)
① uses
Java Code with Node Specifications
④ iterate annotations
Java Annotation Processor (DSL Implementation)
② compiles
③ calls
⑤ generates
Java Compiler (javac, ejc)
⑥ compiles
Generated Java Code for Specialized Nodes
⑦ generates
Executable

Figure 9.2: An overview of the development and build process of a Truffle-based dynamic language runtime adapted from [Hum+14, Fig.8].

9.1.2 DSL and Compiler for High-Performance Dynamic Language Runtimes

Usually, high-performance dynamic language runtimes require a custom VM and compiler both implemented using low-level programming languages increasing implementation and maintenance efforts. Reducing implementation efforts using high-level languages results in poorer performance. Truffle [Wür14] is a Java-based implementation framework providing DSL and execution model to implement *self-modifying Abstract Syntax Tree (AST) interpreters* [Hum+14]. Combined with the Partial Evaluation capabilities of Graal peak-performance compared to state-of-the-art implementations may be reached [Wür+17].

Figure 9.2 provides an overview of the development and build process of a Truffle-based (dynamic) language runtime. The simplest way to implement an interpreter is to add an `execute` method to each AST node. The interpreter could execute the program by traversing the AST executing each node's `execute` method. The Truffle DSL builds upon this idea but provides domain-specific annotations via Java annotations, a mechanism where annotated code is pre-processed by transforming or generating

[30]This is possible due to the ability to access VM data structures via JVM Compiler Interface (JVMCI) [Ros19].

Listing 9.23 A small example using Truffle DSL annotations.

```java
@NodeChild("leftNode") @NodeChild("rightNode")
public abstract class AddNode extends Node {
  @Specialization(rewriteOn = ArithmeticException.class)
  protected int addInts(int leftValue, int rightValue) {
      return Math.addExact(leftValue, rightValue);
  }

  @Specialization(replaces = "addInts")
  protected double addDoubles(double leftValue, double rightValue) {
      return leftValue + rightValue;
  }
}
```

derived code [Sea16]. In Listing 9.23 an example AST node which implements the addition of two numbers is shown. Annotations such as `@NodeChild` declare child nodes of the current defined node. Methods annotated as `@Specialization` can define specific implementations that assume a specific state. Those specialization methods at least require arguments for each declared child in its specialized form. For example, if the addends are of integer type we may use fast integer addition and specialize the node's `execute` method to `addInts(int, int)`. In case the assumption is violated an exception is thrown which will trigger a rewrite to the next, more broad specialization which is `addDoubles(double, double)`. `addInts(int, int)` will be flagged as not appropriate for that note and will not use that specialization again.

Listing 9.24 shows an excerpt of the code that is generated from Listing 9.23. Truffle uses a bitset represented as an `int` to capture the current *state* the node is specialized to. Whenever Truffle rewrites a node's specialization because of the violation of the specialization's constraints the specialization will be excluded. The generic case `executeGeneric()` forwards execution to the respective special case using a bitmask (i.e., `0b1`, `0b10`). The bitmask is Partial Evaluation friendly allowing to prune the other cases. The generated code uses the annotation `@CompilationFinal` to tell the compiler that the field's content might change but may be treated as final when the node is specialized. This marks the field as being a member of the static input to the partial evaluator (cf. Chapter 4). Each change in the field's content must be preceded by the compiler directive `transferToInterpreterAndInvalidate` which exactly does what it is named after; returning from compiled mode to the interpreter and invalidating the

Listing 9.24 Excerpt of the partial evaluation friendly generated code from Listing 9.23.

```
@GeneratedBy(AddNode.class)
public final class AddNodeGen extends AddNode {
  @Child private Node leftNode_; @Child private Node rightNode_;
  @CompilationFinal private int state_;
  @CompilationFinal private int exclude_;
  @Override
  public int executeInt(VirtualFrame frameValue)
    throws UnexpectedResultException {
    int state = state_; int leftNodeValue_;
    try {
      leftNodeValue_ = this.leftNode_.executeInt(frameValue);
    } catch (UnexpectedResultException ex) {
      Object rightNodeValue = this.rightNode_.executeGeneric(frameValue);
      return expectInteger(
        executeAndSpecialize(ex.getResult(), rightNodeValue));
    }
    int rightNodeValue_;
    ... /* repeat for rightNodeValue_ */
    if ((state & 0b1) != 0 /* is-active addInts(int, int) */) {
      try {
        return addInts(leftNodeValue_, rightNodeValue_);
      } catch (ArithmeticException ex) {
        // implicit transferToInterpreterAndInvalidate()
        ...
      }
    }
    CompilerDirectives.transferToInterpreterAndInvalidate();
    return expectInteger(
      executeAndSpecialize(leftNodeValue_, rightNodeValue_));
  }
  ... /* left out other generated code for brevity */
  @Override
  public Object executeGeneric(VirtualFrame frameValue) {
    int state = state_;
    if ((state & 0b10) == 0 /* only-active addInts(int, int) */
    && state != 0
    /* is-not addInts(int, int) && addDoubles(double, double) */) {
      return executeGeneric_int_int0(frameValue, state);
    } else if ((state & 0b1) == 0
    /* only-active addDoubles(double, double) */ && state != 0
    /* is-not addInts(int, int) && addDoubles(double, double) */) {
      return executeGeneric_double_double1(frameValue, state);
    } else {
      return executeGeneric_generic2(frameValue, state);
    }
  }
}
```

Listing 9.25 The bytecode dispatch node of an AST interpreter of a stack-based VM (adapted from [Gri+17, Listing 4].

```
int bci = 1;
while (bci != -1) {
  int next = bcNodes[bci].execute(frame);
  bci = bcNodes[bci].successors[next];
}
```

current node's specialization while updating the state of the node.

During the execution of the interpreter type feedback is gathered and used to rewrite, i.e., to specialize the AST resulting in specialized code being execute. As we have seen in Listing 9.24 the generated code is especially amenable to partial evaluation capabilities of the JIT compiler. As discussed in [JGS94] about the *offline partial evaluator* we know that binding time analysis is easier when *annotations* are provided that mark each parameter, operation, and function call to either be eliminable or residual. To achieve this Truffle provides the aforementioned annotations regarding children, compilation final fields, and specializations. To constrain the size of the code that is partially evaluated another annotation is provided by Truffle. The annotation `@PEBoundary` is used to mark the *end* for the partial evaluator [Wür+17]. The idea to use first Futamura Projection [Fut99] is not new and has been tried for aspect-oriented semantics [MKD03]. As explained in Chapter 4 Partial Evaluation can be implemented, for example, by abstract interpretation of the program. The partial evaluator of Graal generates the Graal IR of the specialized interpreter code during partial evaluation, effectively applying the first Futamura projection [Fut99] (see Chapter 4). The resulting code is further optimized and compiled by the Graal compiler to high-performance machine code.

9.1.3 Meta-circular Java Bytecode Interpreter for the GraalVM

The Graal compiler normally generates Graal IR from Java bytecode. In collaboration with Truffle the Graal IR is generated by Graal's partial evaluator which partially evaluates Truffle-based AST interpreters. TruffleBC [Gri+17] and Espresso[31] [Ora23b] are Truffle-based AST interpreter implementations of the stack-based JVM execution model. This is achieved by representing every bytecode as an AST node. The class `BytecodeNode` implements a bytecode *interpreter loop* of a method's bytecodes

[31]Espresso is a functional prototypical JVM implementation fulfilling the JVM specification.

as shown in Listing 9.25. For each bytecode the stack effect is either executed immediately or AST nodes implement more complex bytecodes. For example, a jump bytecode may directly move the stack pointer to the jump target while an invocation node must lookup the object's method table. Each operation returns the index of the top of the stack which is stored as a local variable, thus rendering the implementation more friendly to partial evaluation. Local variables and method parameters are passed via instances of the class `Frame`. Partial evaluation and escape analysis optimize access to these frame objects [SWM14]. To improve performance TruffleBC unrolls the interpreter loop. Espresso applies *quickening* [Bru10] to AST nodes where general bytecode nodes are *replaced* by more specific AST nodes using the feedback gathered during interpretation. For example, the interpreter can replace a virtual method call bytecode, e.g., `invokevirtual`, with a `DirectInvoke` node. This knowledge can be gathered because Espresso reimplements JVM internal data structures such as classes and functions and their meta-data as Truffle-based language runtime objects. This allows, for example, to optimize function invocations of classes without subclasses more aggressive than classes with subclasses. The partial evaluator or JIT compiler may subsequently inline the method body if there is currently only one implementation available for it [DWM14].

9.2 A Virtual Machine Architecture for Contextual Roles

This section presents essential primitives to enable VM support for contextual roles. We implement these primitives in Espresso for the contextual role-oriented programming language ObjectTeams/Java. Thus, we present ObjectTeams/Truffle, a VM implementation of contextual roles based on the ObjectTeams model [Her07]. While we specifically discuss our approach in the context of the contextual role-oriented programming model ObjectTeams we are confident that our findings can be applied to other role-oriented, class-based programming languages. We start by introducing the essential primitives a VM needs to support contextual roles. Then we discuss how these primitives can be implemented in Espresso, a meta-circular Java bytecode interpreter for the GraalVM [Wür+13].

9.2.1 Essential Primitives to Enable Contextual Roles in a VM

In Chapter 7 we presented how the representation of contextual role-oriented concepts and the implementation of its dispatch semantics imposes a challenge to state-of-the-art VMs. To support contextual roles the components of the VM must be equipped with role-specific primitives. This requires an extension that crosscuts many components such that the VM will be able to apply new optimizations due to these primitives. In the following we will introduce the primitives that we identified as essential to achieve VM support for contextual roles.

9.2.1.1 Dynamic Binding

In Section 3.2 we described that roles change the behavior of role-playing objects. This change in behavior is only constrained by the role model of the application. This is achieved by not only applying the binding analysis at runtime but also to re-evaluate bindings whenever objects play or abandon roles. In Section 3.1.1 we explained that approaches based on AOP provide *pointcut languages*—declarative languages that describe sets of points in the program—where each valid point in the program is marked as *join point shadows*. ROP provides pointcut languages with different expressiveness, too. Furthermore, do contexts group roles, maintaining the status of multiple sets of join points shadows.

The programming model of contextual roles in ObjectTeams is inspired by aspect-oriented concepts. In consequence it must be ensured that all relevant points in the application are found and observed, i.e., the points in the program that must be woven. There are two fundamental ways of weaving join points: The first approach is to weave all eligible call sites independently. A second way is to create an envelope method [Boc+05] to which all eligible call sites delegate to, weaving will take place inside the envelope method. To support roles, a VM must provide support for join point shadows, facilities to weave instructions in the program code, and to issue the re-binding of affected call sites.

9.2.1.2 Meta-Objects for Roles and the Plays Relation

An object may simultaneously play roles in multiple contexts. The activation and deactivation of contexts could change the order in which role instances are played [Küh+15] impacting the observed behavior of the role-playing object itself (cf. Definition 8.3). In state-of-the-art role-based runtimes this relation is transparent to the VM and handled by language runtimes as library implementations. For example, in OTJ the *plays* relation is stored in the meta-data section of class files and loaded at runtime. The MOP of OTJ represents that information at runtime.

The relation must be promoted to a first-class VM citizen. Changes to this relation must be observable by the VM. To support efficient access to and retrieval from the *plays* relation requires additional first-class support for roles and their relations inside the components of the VM.

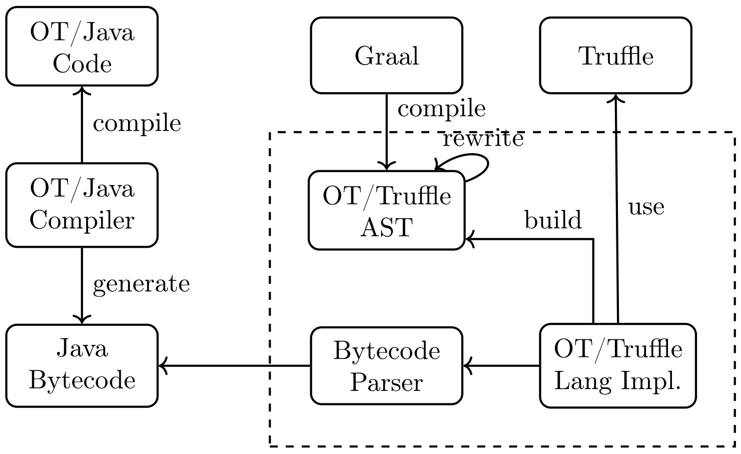


Figure 9.3: Toplevel Architecture of ObjectTeams/Truffle (adapted from [SC23]).

9.2.1.3 Role Dispatch

To the best of our knowledge, since the advance of contextual roles, all programming language implementations for contextual roles fall back to delegation-based implementations to realize role dispatch. We identified multiple reasons for the lack of varying implementation strategies. First, the representation of contexts in class-based languages often is realized by encapsulation. Context classes enclose the role classes, i.e., in Java role classes often are *inner classes* of their enclosing contexts. Second, the access to the *plays* relation requires multiple indirections which increases the cost of resolving role-based call sites. Third, join points are evaluated repeatedly at run-time. The VM must use role-related attributes to generate role-specific code and to provide role-specific optimizations and use internal invalidation mechanism when assumptions do not hold.

9.2.2 Role Support in a Meta-circular Java Bytecode Interpreter

This section gives an overview how we realized the aforementioned fundamental requirements to provide support for contextual roles. The following sections will discuss selected topics more in depth. We based our approach on Espresso, a meta-circular JVM implemented as a Truffle-based AST interpreter. To support the semantics of contextual roles we had make cross-cutting changes to several components of Espresso. In Fig. 9.3 we show the top-level diagram picturing the components of our prototypical approach. Due to Espresso interpreting Java bytecode we kept the two-stage compila-

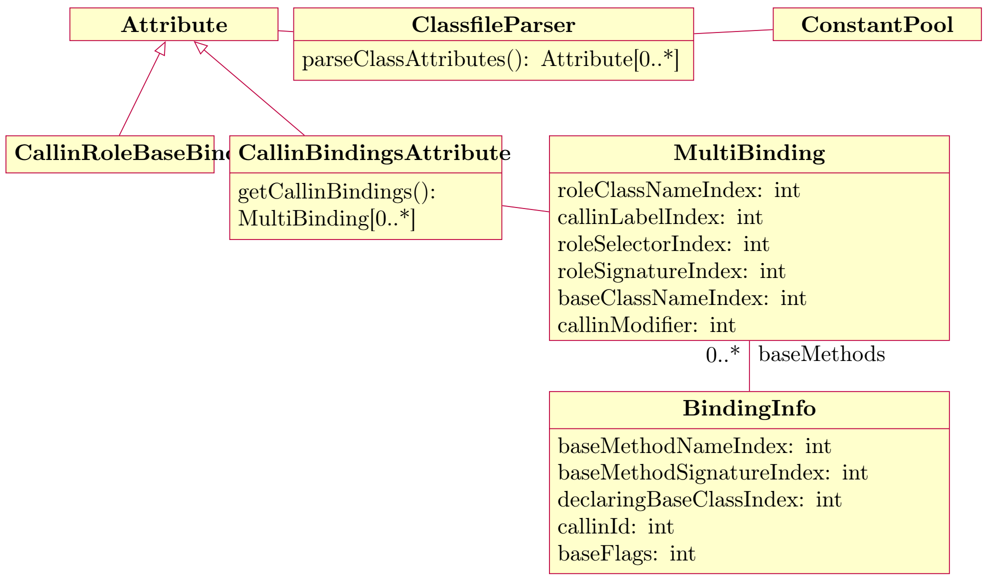


Figure 9.4: ObjectTeams/Truffle classes related to resolving the callin binding attributes parsed from the meta-data section of the class files.

tion process where the OTJ compiler parses and compiles each unit under compilation producing Java bytecode which is the input to the JVM.

To handle join points in Espresso without changing the programming model of OTJ we changed the way the ObjectTeams/Java compiler generates envelope methods. Normally, the body of an original method is rewritten to delegate to the envelope method which puts OTJ internal data on the stack. The original method that triggered the call is not directly accessible anymore and requires an expensive stack walk to recover. Thus, the original method body is changed to put the arguments on the stack just like the envelope did and to invoke a newly defined intrinsic. At runtime when methods are parsed and call sites resolved inside Espresso the intrinsic is used to recover the original method that triggered the call.

The OTJ compiler writes the resolved role bindings (cf. Figure 7.1 depicting the Metaobject Protocol of OTJ) to the attributes section of team classes. We extended Espresso's Java bytecode parser to read the non-standard OTJ-related attributes and to store them in internal VM class representations. The responsible classes and their relation is shown in Figure 9.4. After the VM parsed the bytecodes the ObjectTeams/Truffle (OT/Truffle) language implementation, a component responsible with coordinating the actions of the VM, will build the corresponding AST. In the process of resolving call sites, whenever we discovered a bound call site the VM processes the intrinsic envelope method. Call sites that reference envelope methods will be *quickened* into intrinsic AST nodes that execute role dispatch. This way the VM is able to represent the OTJ runtime model directly with VM data structures instead of using a language level MOP implementations and chains of delegation.

A side effect of having low-level access to VM internal data is that our language implementation can store references to class and method names efficiently by only storing the indices of the strings inside the constant pool of the respective class file. This enables to reuse core VM facilities for method lookup and dispatch. In OTJ all information inside the MOP related to class names, method names, and join points were represented as strings.

In the model of OTJ *lifting* [HHM11, §2.3] realizes the cast-like approach of retrieving the roles played by an object in the current context. To implement lifting we represent the access to the plays relation, represented by the binding information, in a separate AST node. This makes it possible to capture and specialize on values of the domain, i.e., the particular context type that is accessed. The resulting node may be shared among multiple identical liftings that occur during a dispatch.

Our prototype ObjectTeams/Truffle supports all the different kinds of callin bindings, i.e., role methods that are invoked before, after, or as replacement of the original method. The dispatch is realized with multiple AST nodes that implement different parts of the invocation process. The implementation leverages the internal attribute representations to compute the amount of role methods contributed by a respective context and the precedence of the role methods. This allows to unroll loops generating a concrete sequence of instructions in the compiled code of these nodes instead of recursively evaluating the stack. For example, for AST nodes that realize the dispatch to before and after role methods, respectively, the partial evaluator can unroll the loop over contributed role methods since the bounds are known at evaluation time.

9.2.3 Quickening of Role Invocations

ObjectTeams/Java applies the envelope approach [Boc+05] to execute role methods from join points, i.e., base methods that are bound by active callins. The envelopes are the entry point to structured role dispatch and their interfaces are shown in Listing 7.17. To initialize role dispatch OTJ weaves the code of the envelope method `callAllBindings`. The envelope method has the task to collect the relevant runtime values that will be used in the dispatch logic to find the call targets and to put the current runtime state on the stack. The weaver originally statically captured the base method that initiated the call from the lexical scope during weaving, which is required in order to query the runtime for activated contexts (teams) specific to that base method. Instead of generating role-related bytecodes the implementation of ObjectTeams/Truffle quickens the call to the envelope method into role dispatch AST nodes. Figure 9.5 shows a UML class diagram that highlights how we embedded the quickening into Espresso. Our VM implementation quickens the `invokevirtual` bytecode node to overcome the semantic gap, replacing each intrinsic method with the respective AST nodes. These nodes encode the role dispatch semantics usually found in envelopes *callAllBindings* and *callNext* plus optimizations not possible before. This is the entry point to the self-modifying AST that happens during execution.

During the processing of the bytecodes of the base method the interpreter resolves each `invoke` bytecode and replaces it with the appropriate AST dispatch node. Listing 9.26 shows an excerpt of the method that implements the replacement for the cases of `callAllBindings` and `callNext`. The first case (lines 1–3) resolves `callAllBindings` and has lexical access to the base method which makes quickening easy. During the execution of the quickened node we get access to the stack frames and install the base method using a magic token. The second case concerning `callNext`, i.e., to *proceed*, evaluating a `base` call, from a replace callin is more involved (see lines 4 – 23). Proceed is naturally called from within the body of a role method. The lexical scope of proceed is not able to capture the method that initiated the role dispatch. To compute the base method that caused the activation we walk the stack of activation records (i.e., stack frames) which can be accessed as first-class entities through visitor facilities provided at VM level (see lines 7–18). We can extract the base method using the magic token that was installed when executing the quickened `callAllBindings` node.

9.2.4 A Self-Modifying AST to Represent Contextual Role Dispatch

We postulate that efficient dispatch is key for performance. In recent works it has been shown that contextual role dispatch could be optimized by applying partial evaluation [SKC22]. This can be achieved by representing the dispatch using primitives that can be understood and optimized by a language runtime. However, we observed that the results after partial evaluation is a too coarse-grained and rigid structure that requires new computations and partial evaluations whenever contexts change.

As described in Section 9.2.3 the implementation of ObjectTeams/Truffle provides VM implementations for essential primitives of contextual roles. The role dispatch is implemented with several AST nodes providing general and optimized code paths, i.e., using *assumptions* and specializations to declare special application states. In Figure 9.6 we show part of an AST that replaces the envelope *callAllBindings* that delegates to the replace callin `CheckingsAccount.withFee(*)` woven for `acc.withdraw(*)`

Listing 9.26 Quickening `invokevirtual` bytecodes to role invocation AST nodes. The listing shows an excerpt of the method `dispatchQuickened(...)` for `callAllBindings` and `callNext`.

```java
} else if (opcode == INVOKEVIRTUAL && resolved.isBoundMethod()) {
  invoke = new InvokeCallAllBindingsQuickNode(
    resolved, this.getMethod(), top, curBCI);
} else if (opcode == INVOKEVIRTUAL && resolved.isCallNext()) {
  // use VirtualFrame auxilary slots to add OTJ magic objects
  CompilerDirectives.transferToInterpreterAndInvalidate();
  FrameInstance iteratedFrame = Truffle.getRuntime().iterateFrames(
    new FrameInstanceVisitor<FrameInstance>() {
    @Override
    public FrameInstance visitFrame(FrameInstance frameInstance) {
      EspressoRootNode rootNode = getContext().getVM()
        .getEspressoRootFromFrame(frameInstance);
      Frame visitedFrame = frameInstance.getFrame(
        FrameInstance.FrameAccess.READ_ONLY);
      Method m = rootNode.readFrameBaseMethodOrNull(visitedFrame);
      return (m == null) ? null : frameInstance;
    }
  });
  Frame otherFrame = iteratedFrame.getFrame(
    FrameInstance.FrameAccess.READ_ONLY);
  EspressoRootNode rootNode = (EspressoRootNode) getRootNode();
  Method baseMethod = rootNode.readFrameBaseMethodOrNull(otherFrame);
  invoke = new InvokeCallNextQuickNode(resolved, baseMethod, top, curBCI);
} else { ... }
return invoke;
```

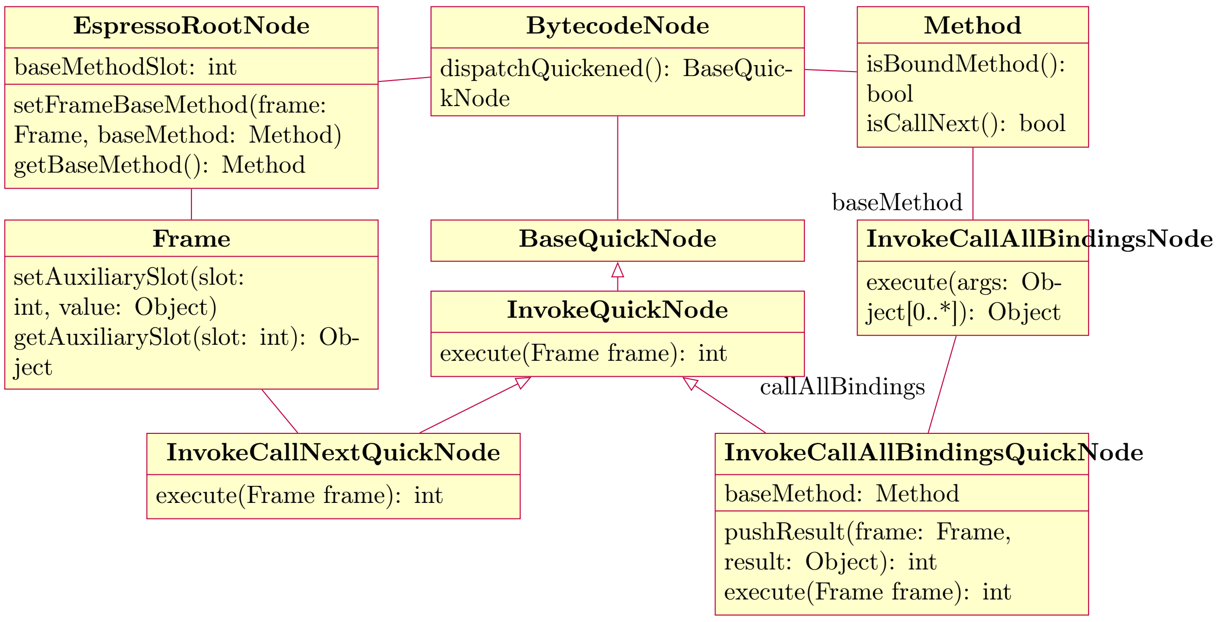

EspressoRootNode
baseMethodSlot: int
setFrameBaseMethod(frame: Frame, baseMethod: Method)
getBaseMethod(): Method
BytecodeNode
dispatchQuickened(): BaseQuickNode
Method
isBoundMethod(): bool
isCallNext(): bool
Frame
setAuxiliarySlot(slot: int, value: Object)
getAuxiliarySlot(slot: int): Object
BaseQuickNode
baseMethod
InvokeCallAllBindingsNode
execute(args: Object[0..*]): Object
InvokeQuickNode
execute(Frame frame): int
callAllBindings
InvokeCallNextQuickNode
execute(Frame frame): int
InvokeCallAllBindingsQuickNode
baseMethod: Method
pushResult(frame: Frame, result: Object): int
execute(Frame frame): int

Figure 9.5: A UML class diagram highlighting key changes to Espresso to implement ObjectTeams/Truffle and the quickening of bytecodes into AST nodes to realize role dispatch semantics.

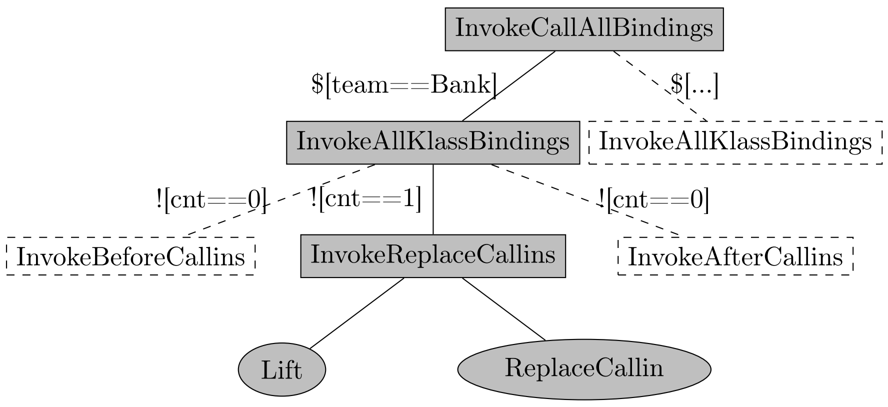


Figure 9.6: The intrinsic AST that replaces the envelope *callAllBindings* that delegates to the replace callin `CheckingsAccount.withFee(*)` woven for `acc.withdraw(*)` in Listing 6.12. Rectangles represent nodes, dashed means the node will not be materialized. Ellipses represent call targets. The edge labels `![...]` declare an assumption and `$[...]` declare a cached value.

in Listing 6.12. The rectangles represent AST nodes. The edge labels `![...]` declare an assumption and `$[...]` declare a cached value. Dashed elements means the node or edge (implemented via references) will not be instantiated. This may happen whenever a node violates an assumption or a guard. Ellipses represent call targets such as the callin of the role.

We want to highlight that we decided to not evaluate the whole runtime state at once and guard the resulting callable by the partial structural equality of runtime state (cf. Definition 8.8). Instead, the beginning of role dispatch is represented by nodes that are guarded by the most recent context type of the runtime state, e.g., the team `Bank`. For example, if there is no active context the role dispatch will not be executed at all but the original method. In subsequent calls a guard checks whether the first active context is of a different type and either reuses the node or embeds a new one, effectively creating a PIC [HCU91] for contexts. It also enables the possibility to only change parts of the AST instead of having to recompute the overall dispatch when the application state changes. Represented in Figure 9.6 is the usage of assumptions to efficiently prevent the creation of unused subtrees for non-existent callins. For example, when the `InvokeBeforeCallins` node is created the VM reads the internally stored attributes of contexts and their bindings and toggles the assumptions about the contributed role methods accordingly. A context can declare multiple roles each with multiple before and after bindings for the same base method. In such a case all role methods are called before and after, respectively, ordered by their declared precedence.

On the other hand we can optimize the evaluation for each callin type of the context. For example, as shown in Listing 9.27 the evaluation loops over the statically known length of before callins for a specific base method. The annotation `@ExplodeLoop` instructs the partial evaluator to unroll the loop. This produces a sequence of instructions that is easier to optimize by the JIT compiler. Lifting calls the OTJ internal functions for object management and invokes the role function on the returned object. This requires proper handling of the arguments and potential insertion of casts from and to Truffle's API, for example, `StaticObject` for Java's `Object` of any kind.

9.3 Performance Evaluation

This section evaluates the run-time performance and characteristics of ObjectTeams/Graal, a VM implementation of ObjectTeams. We compare our approach to the reference implementation ObjectTeams/Java [Her07] and ObjectTeams with Dispatch Plans [SKC22] (ObjectTeams/InvokeDynamic).

9.3.1 Benchmark Characterization

We based the benchmark on the synthetic benchmark introduced in Section 6.2. To evaluate different characteristics of context-dependent software we used the *dynamic* use-case with variable context activations.

9.3.2 Methodology

We measured the execution time per run and report the geometric mean and standard deviation for each problem size and approach as discussed in Section 6.2.2. To observe whether there are scalability problems each approach is executed with multiple problem sizes varying the problem size from 1.0 to 2.5 million transactions. Because of the increased execution time every benchmark we deviated from Section 6.2.2 and ran the benchmark 5 times for each data point.

To coordinate the benchmark execution and to allow proper measurement of the experiments we used the benchmark execution framework ReBench [Mar18]. The benchmark was conducted on an Intel Core i7-9700T CPU @ 2.00GHz with 32GB RAM. For the evaluation we built Espresso from commit 393e30fb1b9 with mx version 6.19.0. Our VM build is based on LabsJDK CE17 JVMCI v23.0b01.

Listing 9.27 The implementation of evaluating before callins of a team in ObjectTeams/Truffle.

```
@ExplodeLoop
@Specialization
Object doAllBefores(Object[] args, StaticObject teams,
  @Bind("getIndex(args)") int index,
  @Bind("getCallinIds(args)") StaticObject callinIds,
  @Bind("getOriginalArgs(args)") StaticObject originalArgs,
  @Bind("getBoundBase(args)") StaticObject boundBase) {
  /*
   * The int[] callinIds is actually already sorted before/after/replace.
   * We already know the team klass, the bindings and their precedence and order.
   */
   final Language lang = getLanguage();
   for (int i = 0; i < length; i++) {
     final int updatedIndex = scatterIndeces[i] + index;
     StaticObject team = teams.get(lang, updatedIndex);
     final int callinId = callinIds.<int[]>unwrap(lang)[updatedIndex];
     MultiBinding binding = Utils.getBindingForId(bindings, callinId,
       CallinBindingsAttribute.KIND_BEFORE);

     if (binding == null) {
       return Boolean.FALSE;
     }

     final int roleClassNameIndex = binding.getRoleClassNameIndex();
     Method liftMethod =
       Utils.lookupLiftMethod(teamKlass, roleClassNameIndex);
     DirectCallNode lift =
       DirectCallNode.create(liftMethod.getCallTarget());
     StaticObject roleObject = (StaticObject) lift.call(team, boundBase);

     Method roleMethod = Utils.lookupRoleMethod(teamKlass,
       (ObjectKlass) roleObject.getKlass(), binding);
     DirectCallNode roleNode =
       DirectCallNode.create(roleMethod.getCallTarget());
     Object[] roleCallArgs = new Object[originalArgs.length(lang) + 1];
     roleCallArgs[0] = roleObject;
     for (int l = 0; l < originalArgs.length(lang); l++) {
       StaticObject obj = originalArgs.get(lang, l);
       roleCallArgs[l + 1] = getMeta().unboxFloat(obj);
     }
     roleNode.call(roleCallArgs);
  }
  return Boolean.TRUE;
}
```

9.3.3 Performance Analysis

The results of the dynamic case are shown in Figure 9.7. The figure depicts the geometric mean in seconds for each approach and each problem size.

From the results we see that for ObjectTeams/Java and ObjectTeams/Truffle the standard deviation increases with increasing problem size and duration. We assume the reason is that these approaches cause longer garbage collection pauses and use more of the assigned heap. This may be caused by the handling of runtime values during the execution of the dispatch. The standard deviation for ObjectTeams/InvokeDynamic is lower across all problem sizes. We assume the reason is the implemented copy-on-write optimization for the data structure representing the active team instances. Such an optimization is not implemented in ObjectTeams/Truffle.

From the conducted experiment we report that ObjectTeams/Java performs as the slowest implementation. The role-agnostic VM implementation ObjectTeams/Truffle reaches a speedup of up to 2.49× (mean 2.23×) compared to ObjectTeams/Java. Compared to ObjectTeams/InvokeDynamic the approach is up to 1.22× faster (mean 1.18×).

9.3.4 Threats to Validity

We are aware that Espresso is still an early prototype of a meta-circular Java VM. It passes the Java Compatibility Kit but might have unequally good implementations for different parts of the JVM specification. This could skew the results and improperly favor one implementation over the other. However, even under these conditions, we are confident that the results of this work are meaningful enough.

The reference implementation ObjectTeams/Truffle supports a limited set of features provided by ObjectTeams/Java. It is possible that the introduction of other features may require additional checks or rewrites of the AST imposing restrictions on the implementation we did not account for, yet. The benchmark purposely avoided features that have not been implemented across all compared implementations of the ObjectTeams model to ensure comparability.

9.4 Conclusion

This chapter presented that the first Futamura Projection [Fut99] can create efficient interpreters and the optimization potential introduced by combining Partial Evaluation and Just-In-Time compilation. We introduced how a state-of-the-art VM can be

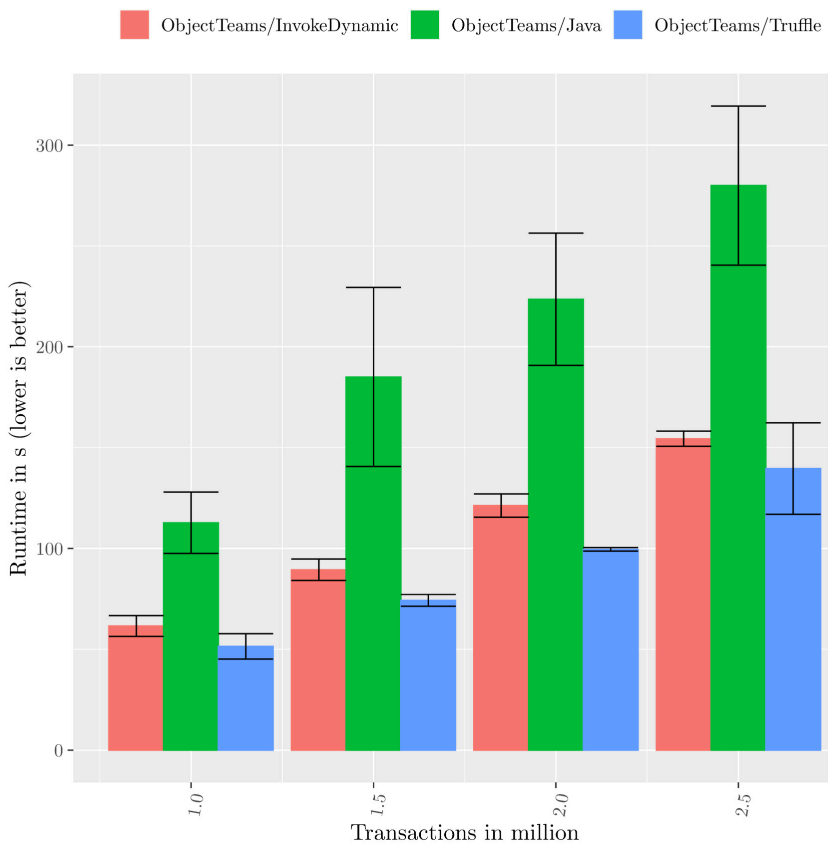


Figure 9.7: Execution time in seconds for the dynamic case. The error bar shows the standard deviation of the measured run-time.

extended with essential primitives required to optimize contextual role-oriented programming languages. The combination of a role-aware AST, and an partial evaluation friendly interpreter reduced the semantic gap. For example by annotating *compile-time* static values, by annotating parameters used to construct Polymorphic Inline Caches, and declarative guards used to decide when to reuse a value of a parameter. The result after PE is a smaller IR with more opportunities for optimization by the Graal JIT compiler.

Since VMs for object-oriented programming languages do not understand the semantics of contextual roles, the compiler produces a verbose description of roles in an object-oriented paradigm, which incurs a high overhead. We proposed essential primitives to support roles in a VM. We present a prototypical VM implementation to efficiently execute contextual roles. While we specifically discuss our approach in the context of the contextual role-oriented programming language ObjectTeams we are confident that our findings can be applied to other role-oriented, class-based programming languages. For a demanding role-based benchmark we achieved a speedup of up to 2.49× compared to the reference implementation ObjectTeams/Java. Compared to ObjectTeams with dispatch plans [SKC22] our approach achieves a speedup of up to 1.22×.

Because of their commonalities, we are confident that our approach is also able to work in the context of AOP languages using the pointcut-advice model and layered COP languages. The execution of behaviors in each of these related approaches is concerned with the evaluation of joinpoints and a multi-dimensional dispatch, which could be mapped to the role dispatch discussed in this thesis.

Part IV

Related Work, Conclusion, and Outlook

10 Related Work

"Think left and think right and think low and think high. Oh, the thinks you can think up if only you try!"
— Theodore Seuss Geisel, a.k.a. Dr. Seuss

During the last decades a surmount of approaches have been proposed that may be ascribed to the domain of MDSoC. We introduce the most relevant of these approaches, dividing them in the following three sections according their belonging into the categories of dynamic programming languages and Metaobject Protocols, Aspect-oriented Programming, and Context-oriented Programming. The approaches in each category comprise from additional language constructs and domain-specific compilers to VMs and domain-specific JIT compilers.

10.1 Dynamic Programming Languages and Meta-Object Protocols

Optimizing dispatch is a recurring topic among approaches that utilize the reflective capabilities of host languages or MOPs. MOPs enable the definition of extended semantics in the host language itself [KDB91]. It allows defining domain-specific dispatch semantics from within the host language instead of relying on custom VM or compiler support.

The meta-tracing JIT compiler of PyPy can be controlled by providing runtime feedback in language-specific ways [Bol+11]. Meta-tracing describes that the instruction sequence, i.e., the *trace*, of the interpreter is recorded and not the application executed itself. Language implementers can use *hints* to allow fine-tuning of the JIT compiler decisions. Feedback is considered by describing the dynamic shapes of objects and providing runtime-type information. The optimizations are mainly used to optimize the object model representing the dynamic programming language inside the interpreter. Such an object model to optimize dynamic languages in a method-based JIT compiler can also be found in [Wöß+14].

A call site in a single-dispatched object-oriented programming language (cf. Chapter 2) consists of the name of the function, the argument list, and the type of the callee. The de-facto standard to optimize such a polymorphic call site uses PICs to capture the lookup result for different types of callees improving the performance of future dispatches [HCU91]. Dynamic languages often use reflective capabilities of the VM to implement the flexible dynamic dispatch mechanisms sacrificing performance by doing so. That is because the argument types of a *reflective* call site—including the target's object type as well as the method name—are only known once the actual invocation happens. A single PIC is not designed to capture these cases since same-named methods can be called on *unrelated* objects. To remove the overhead of reflection and Metaobject Protocols, a *dispatch chain* presents a generalization of Polymorphic Inline Caches that allows optimization of metaprogramming at the VM level [MSD15]. A dispatch chain consists of a few linked nodes where the last node resembles an ordinary PIC. Each level can store the result of the intermediate step of the lookup process for further reuse. For example, for the reflective invocation of `calc.invoke('inc, ['once])` a dispatch chain for `invoke()` is created. The first level observes the `'inc` symbol, and the second level then is a classical PIC that maps the symbol `calc` to the type `Calculator` and directly refers to the implementation for `Calculator.invoke(...)`. Dispatch chains can be used with meta-tracing JIT compilers or compilers using partial evaluation and perform equally well for both approaches.

JIT compilers can reduce the performance gap between dynamic and static programming languages tremendously. To successfully speculate the runtime variability must be low and heuristics must be able to determine when optimization is beneficial. However, some variability patterns are hard to capture in a heuristic. Compiler heuristics often do not capture ephemeral, warmup, sporadic, and highly indirect variability. A Metaobject Protocol allows identifying these types of variability at the application level. To communicate variabilities to the VM call sites in the application can be annotated by developers providing additional information to the compiler [CGM17]. The MOP offers application developers resetting compiler-build dispatch chains or issuing a node split reducing the pollution of profiles for each resulting local call site. Thus, speculative optimizations can be issued where heuristics would traditionally fail.

In Chapter 8 we presented a graph-based approach claiming to represent a generalized PICs tuned to capture contextual roles. Our approach did not generalize over single reflective call sites but compositions of functions that may be invoked from a single call site. The variability is already captured in the MOP describing the role semantics. Partially evaluating that MOP results in the dispatch graph already being specialized for the callins that are going to be expected only. Trimming the profile or reverting to the most general case is something that all approaches have in common.

10.2 Aspect-Oriented Programming

The Aspect-oriented Programming concept is a promising approach to many problems encountered in software engineering and is highly related to Role-oriented Programming. However, it experienced a huge performance gap compared to classical object-oriented approaches. The first approaches used basic capabilities of target architectures and languages such as component-based designs using hooks [Paw+01; BH02]. Research approached the problems by proposing AOP specific extensions to object-oriented VMs. The availability of meta-circular research VMs, for example, Jikes Research Virtual Machine (RVM) [Alp+05], MaxineVM [Wim+13], and Graal VM [Dub+13a; Wür14], that is, for example, a JVM for and in Java, fostered stronger research in the domain of VMs. Specifically, many approaches extended the Jikes RVM, an open-source research Virtual Machine [Alp+05] mostly written in Java.

The first approach we introduce here explored to enhance the JIT compiler of the Jikes RVM [PGA02; PAG03]. To support weaving at join points the JIT compiler is extended to compile minimal hooks. These hooks add runtime checks that guard the dispatch to advise code.

To compile and execute AOP programs they must go through many steps that process and alter or generate code (see Section 3.1.1 for a background on AOP). For each point in the program (i.e., join point shadows) that is designated by a pointcut code to dispatch to advice is *woven*. Using join point shadows as weaving locations increases the weaving time as these are, in general, spread all over the program. Envelope-based weaving reduces the weaving time by introducing changes to the callee, not the caller [Boc+05]. To decrease the number of weaving locations accessors for fields and proxies for methods are introduced. These *envelopes* are generated inside the declaring class and the original method body is moved. This simplifies the weaving process a lot and reduces the overall time required to compile aspectual programs.

The indirection introduced by envelopes resulted in an overhead that could not be mitigated completely by inlining from state-of-the-art VMs. To further mitigate the overhead envelope-aware optimizations have been introduced to the Jikes RVM [Boc+06]. One approach is to make the inline oracle used by the optimization to inlining methods always to decide to inline envelopes. On the other hand, the timer-based sampler is changed to credit the sample of an implementation method to its proxy envelope by walking down the stack one additional frame. This increased warmup time until the VM reached a steady state because the proxy envelopes required more rounds of optimizations until being finally optimized away. Inlining became possible due to changes in RVM's guarded inlining and code patching strategies.

Another extension of the Jikes RVM is found within the Steamloom VM [Boc+04].

AOP concepts are represented as first-class entities and advice integration and execution are managed by the VM. The approach is integrated with CaesarJ's [Ara+06] dynamic deployment API and supported thread-local and instance-local deployment of aspects. In contrast to other approaches, Steamloom does not require pre- or post-processing such as running a weaver to integrate aspects. Dynamic weaving is embedded in the VM. The Bytecode Augmentation Toolkit (BAT) is responsible for weaving advice code and uses a custom format to represent an advised method's bytecodes instead of using RVM's `byte` arrays. Steamloom uses super instructions to implement AOP semantics [Hau+05]. The required information is stored in an Advice Instance Table (AIT) a runtime data structure embedded in the memory representation of classes managing aspect objects the JIT compiler uses to generate aspect code into method bodies [HM05]. Each entry of instance-local aspects must be cloned from the class adapted by the aspect. Inlining method bodies of these classes is not realized as multiple versions of these classes— advised and non-advised—exist. A shortcoming of Steamloom is that only `before` and `after` advice for method execution join points can be declared. The authors remark that `around` advice (the pendant to *replace* in role-oriented programming) is a more demanding challenge [HM05].

Our approach is not affected by this limitation. First, roles always represent instance-local aspects because of their close relation to the playing objects. Second, our approach is concerned with the callsites themselves and inlining the dispatch code to role functions. We do not control whether and how inlining is performed by the VM.

Supporting AOP resolves around providing dynamic redefinitions class definitions. However, state-of-the-art production JVM often limits this support. As a consequence approaches elude to runtime weaving of previously loaded classes. The authors redesign a production-grade JVM to offer unrestricted class redefinition and redesign the HotWave AOP framework [Vil+09] to overcome the aforementioned restrictions [Wür+10].

The Nu VM [DR10] is another extension to Java HotSpot VM but only supports the interpreter; the JIT compiler is not supported. The authors propose a new IR to capture and support dynamic AOP features.

10.3 Context-Oriented Programming

COP is another programming paradigm that solves many problems encountered in OOP but experiences a huge performance gap compared to classical object-oriented approaches, too.

In JCOP [AH12b], a Java-based COP language, the lookup, and execution of layered dispatch have been replaced to use the `invokedynamic` bytecode [AHH10]. The `invokedynamic` bytecode is different than other `invoke` bytecodes such that it calls a `bootstrap` method the first time it is executed by the JVM. The `bootstrap` method must return an objectified call site object which is executed on subsequent invocations. The call site links to an actual method using `method handle` that may be understood as typed pointers. The approach uses a map to store call targets to partial methods that are managed by each layer. Each call to `proceed` will link the next partial method of the next active layer stored in that map. When a composition is changed, i.e., a layer is added or removed, the method handle of the call site is updated. Compared to the unmodified implementation of JCOP, they report a speedup of 160× when no layer is activated, as well as a speedup of 48 - 38 × for 1 - 5 layers activated.

ContextJS is a context-oriented extension to JavaScript [Lin+11]. In ContextJS the layered dispatch is optimized by reducing the number of method calls and lookups by gradually inlining partial methods into a generated wrapper function [KLH12]. Initially, the wrapper delegates to each partial method but subsequently inlines the delegatee to reduce the overhead. When the application stabilizes, the partial function boundaries are removed and the method bodies are inlined. To further speed up subsequent invocations active compositions can be cached and guarded to check for no change. While the inlining improves the performance by 10× it still just reaches 3% of the performance of a comparative implementation using the host language only. We suspect the approach to rewriting the AST at runtime to fasten runtime code generation instead of generating new code interferes with VM optimizations.

A common approach to speed up dispatching is to use a cache and invalidate approach. In COP an invalidation happens when a new layer imposition changes the targeted partial method of an already cached call site. ContextPyPy [PFH16] uses the meta-tracing JIT compiler to communicate the variability of COP to the VM via annotations where sideway composition introduces an overhead. By *promoting* context-oriented dispatches the JIT compiler ensures that traces are specialized overriding the result of any heuristic. Thus, the VM can reuse recorded traces when layer compositions are stable. Invalidation is realized with a guarded switch.

In contextual role-oriented programming, compartments introduce highly indirect variability. Our approach encodes opportunities to optimize specific call sites. Layered method dispatch with `invokedynamic` [AHH10] uses the same bytecode as our approach presented in Chapter 8. First, they do not construct a call graph but iterate the composition of layers using stored handles of each partial method from a map. For each partial method, there is its call site object that is installed into the call site whenever layers get activated and deactivated. Invoking a `proceed` returns the next call site object pointing to the next partial method. In contrast, our approach defines the whole execution as a call graph that is optimized by the VM. This approach allows us to install guards and to apply PIC on the role-providing contexts. Without applying compiler directives, we rely on the fact that the abstraction used by dispatch graphs is optimized by the VM. In Chapter 9 we described an implementation of a role-aware VM. We do not only provide the heuristics but also the interpreter logic to efficiently execute role-based programs.

11 Conclusion

This chapter summarizes the contributions made in this thesis and discusses possible directions for future work.

11.1 Summary

This thesis reviewed past and contemporary relational, behavioral, and contextual role-oriented programming languages. The review highlights the different concepts applied to implement role semantics and, when possible, derives statements about the expected run time properties of those programming language implementations.

A quantitative analysis of contemporary contextual role-oriented programming languages makes it possible to determine run-time properties such as run-time performance and substantiates the existence of a *semantic gap*. In Chapter 7 the semantic gap based on a concrete implementation of contextual roles is described. Building on the results of the quantitative analysis and the description of the semantic gap this thesis proposes two methodologies and their proof-of-concept implementation.

This thesis contributes a runtime code generation strategy to generate efficient dispatch code based on a partial evaluation of the runtime data of the program. This includes the definition of an object model for the contextual role-oriented programming language ObjectTeams/Java. The partial evaluator uses the object model to evaluate an invocation of a role-bound call site to a call graph that, when executed, is semantically equivalent to the original program.

To overcome the semantic gap this thesis introduces an extension of a meta-circular Java VM to support contextual roles. In Chapter 9 essential primitives to support contextual roles are enumerated and a VM architecture is presented to provide those primitives. The implementation is based on quickening and optimizes role-based dispatch by defining guards that respect the role semantics resulting in efficient code generated due to the first Futamura projection applied by the VM.

In conclusion, this thesis introduced two methodologies that closed the semantic gap measured in Chapter 6 and further described in Chapter 7. If using a custom VM implementation is undesirable runtime code generation based on partial evaluation provides a particularly well-performing runtime of contextual role-oriented programs. A custom VM implementation even exceeds the performance gain.

11.2 Future Work

While this thesis makes a significant contribution to the implementation techniques for contextual role-oriented programming languages we identified further and new interesting research opportunities for future work.

11.2.1 Mapping Strategies

Kühn et al. proposed a feature model that describes the role features ascribed to role-oriented modeling and programming languages [Küh+14]. In his thesis Graversen describes a feature model to capture the strategies used by role-oriented programming languages [Gra06]. Surveying past and contemporary role-oriented programming languages in Chapter 5 introduces multiple mapping strategies to realize role features and role dispatch. Further research on the relations between role language features and their implementation strategies may create a knowledge base that combines both feature models. This opens interesting opportunities to investigate the characteristics of using different mappings to realize the same role features as future work.

11.2.2 Prototype-based Role-Oriented Programming

The contributions of this thesis towards implementation techniques only touch the *class*-based approaches. Implementation techniques for *prototype*-based approaches are open to future work.

Acquiring and abandoning roles impacts the properties and methods available on an object; it changes the observable type of an object. In general, it is possible to statically represent this dynamic change of types by generating all possible combinations of the object's type and its roles and assigning an object to such a type with each change. In statically compiled languages, however, this results in an exponentially growing class hierarchy which usually is sparsely inhabited at runtime.

Optimizing access to objects in prototype-based languages is often realized using *hidden classes* that represent an object's current available properties and methods. This class-like construct, first introduced in SELF [CUL89], allows the VM to apply PICs to dispatches in prototype-based languages. In prototype-based implementations, role-playing is usually realized by using delegation. Thus, it is possible to create such a class hierarchy on demand by having roles change the hidden class of an object. Dynamically creating these intersection classes using runtime code generation or dynamic compilation supported by the flexibility of prototype-based programming languages is a research direction worth exploring [Kör21].

Similarly, it is to be discussed how the dynamic approach of SCROLL where role-playing is based on structural sub-typing mechanisms and predicates may be represented and optimized using prototype-based programming language runtimes.

11.2.3 Memory Model for Role-Oriented Programming

The discussion of essential primitives in Chapter 9 touches on the requirement for first-class support to represent roles and the plays-relation. While the implementation provides first-class support for the plays-relation inside the VM, the role-aware VM still uses the language-level MOP of OTJ. However, the heap and object graph are immediately accessible within the VM. It is future work to extend the memory model of the VM and to introduce new VM classes to manage the role-playing relation. For example, having the VM handle registering the play-relation whenever a context activation and deactivation happens could make handling those relations on the language-level MOP obsolete.

List of Figures

List of Tables

List of Listings

Acronyms

AO	aspect-oriented	
AOP	Aspect-oriented Programming	11
API	Application Programming Interface	77
AST	Abstract Syntax Tree	152
ACL	Agent Communication Languages	77
AIT	Advice Instance Table	176
BAT	Bytecode Augmentation Toolkit	176
COP	Context-oriented Programming	11
CLOS	Common Lisp Object System	17
CROM	Compartment Role Object Model	20
CROI	Compartment Role Object Instance Model	28
CoV	coefficient of variation	90
DAG	Directed Acyclic Graph	52
DSL	Domain-specific Language	11
DBMS	Database Management System	49
DOOR	Dynamic Object-Oriented database programming language with Roles	59
FEJ	Featherweight EpsilonJ	79
FIPA	Foundation for Intelligent Physical Agents	77
IR	Intermediate Representation	134
JVM	Java Virtual Machine	21
JS	JavaScript	29
JIT	Just-In-Time	86
JMH	Java Microbenchmark Harness	89
JNI	Java Native Interface	130

JVMTI	Java Virtual Machine Tooling Interface	106
JSR	Java Specification Request	129
JVMCI	JVM Compiler Interface	152
JAWIRO	Java With Roles	71
LAC	Lua Aspectual Components	75
MDSoC	Multi-Dimensional Separation of Concerns	11
MDSD	Model-driven Software Development	5
MOP	Metaobject Protocol	17
OT	ObjectTeams	32
OTJ	ObjectTeams/Java	32
OO	object-oriented	4
OOP	Object-oriented Programming	6
OOA	Object-oriented Analysis	4
OOD	Object-oriented Design	4
PIC	Polymorphic Inline Cache	12
PE	Partial Evaluation	35
PHAM	Persistent Hierarchical Abstract Machine	52
ROP	Role-oriented Programming	6
$\mathcal{RICA}$	Role/Interaction/Communicative Action	77
RVM	Research Virtual Machine	175
SoC	Separation of Concerns	19
SOP	Subject-oriented Programming	14
SCROLL	Scala ROles Language	29
SSA	Static Single Assignment	130
SUM	Single Underlying Model	98
SoN	Sea-of-Nodes	151
TLAB	Thread Local Allocation Buffer	102
UML	Unified Modeling Language	4
VM	Virtual Machine	7

Bibliography

[ABV92] Mehmet AkŞit, Lodewijk Bergmans, and Sinan Vural. “An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach”. In: *ECOOP ’92 European Conference on Object-Oriented Programming*. Ed. by Ole Lehrmann Madsen. Vol. 615. Berlin/Heidelberg: Springer-Verlag, 1992, pp. 372–395.

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini. “GALILEO: A Strongly-Typed, Interactive Conceptual Language”. In: *ACM Transactions on Database Systems* 10.2 (June 1985), pp. 230–260.

[AGO88] A. Albano, G. Ghelli, and R. Orsini. “The Implementation of Galileo’s Persistent Values”. In: *Data Types and Persistence*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 253–263.

[AGO93] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. “An Object Data Model with Roles”. In: *Proceedings of the 19th VLDB Conference*. Dublin, Ireland, 1993.

[AGO95] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. “Fibonacci: A Programming Language for Object Databases”. In: *The VLDB Journal* 4.3 (July 1995), pp. 403–444.

[AH12a] Malte Appeltauer and Robert Hirschfeld. “Declarative Layer Composition in Framework-Based Environments”. In: *Proceedings of the International Workshop on Context-Oriented Programming - COP ’12*. The International Workshop. Beijing, China: ACM Press, 2012, pp. 1–6.

[AH12b] Malte Appeltauer and Robert Hirschfeld. *The JCop Language Specification: Version 1.0*. Technische Berichte Des Hasso-Plattner-Instituts Für Softwaresystemtechnik an Der Universität Potsdam 59. Potsdam: Universitätsverlag Potsdam, 2012. 48 pp.

[AHH10] Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. “Layered Method Dispatch with INVOKEDYNAMIC: An Implementation Study”. In: ACM Press, 2010, pp. 1–6.

[AHL13] Malte Appeltauer, Robert Hirschfeld, and Jens Lincke. “Declarative Layer Composition with The JCop Programming Language.” In: *The Journal of Object Technology* 12.2 (2013), 4:1.

[Ald20] Alessandro Aldini. “Quantitative Aspects of Programming Languages and Systems over the Past 2^4 Years and Beyond”. In: *Electronic Proceedings in Theoretical Computer Science* 312 (Jan. 20, 2020), pp. 1–19.

[Alp+05] B. Alpern et al. “The Jikes Research Virtual Machine Project: Building an Open-Source Research Community”. In: *IBM Systems Journal* 44.2 (2005), pp. 399–417.

[ALS08] S. Apel, T. Leich, and G. Saake. “Aspectual Feature Modules”. In: *IEEE Transactions on Software Engineering* 34.2 (Mar. 2008), pp. 162–180.

[And11] Marc Andreessen. “Why Software Is Eating The World”. In: *The Wall Street Journal* (Aug. 20, 2011).

[App+09] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. “A Comparison of Context-Oriented Programming Languages”. In: International Workshop on Context-Oriented Programming. ACM Press, 2009, pp. 1–6.

[App+10] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. “Event-Specific Software Composition in Context-Oriented Programming”. In: *Software Composition.* Vol. 6144. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 50–65.

[AR92] Egil P. Andersen and Trygve Reenskaug. “System Design by Composing Structures of Interacting Objects”. In: *ECOOP ’92 European Conference on Object-Oriented Programming.* Vol. 615. Berlin/Heidelberg: Springer-Verlag, 1992, pp. 133–152.

[Ara+06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. “An Overview of CaesarJ”. In: *Transactions on Aspect-Oriented Software Development I.* Ed. by Awais Rashid and Mehmet Aksit. Vol. 3880. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 135–173.

[Aßm03] Uwe Aßmann. *Invasive Software Composition.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[BA01] Lodewijk Bergmans and Mehmet Aksit. “Composing Crosscutting Concerns Using Composition Filters”. In: *Communications of the ACM* 44.10 (2001), pp. 51–57.

[BA04] Lodewijk Bergmans and Mehmet Aksit. “Principles and Design Rationale of Composition Filters”. In: *Aspect-Oriented Software Development* (2004), pp. 63–95.

[BA12] Fernando Sérgio Barbosa and Ademar Aguiar. “Modeling and Programming with Roles: Introducing JavaStage”. In: *Frontiers in Artificial Intelligence and Applications* (2012), pp. 124–145.

[Bac73] Charles W. Bachman. “The Programmer as Navigator”. In: *Communications of the ACM* 16.11 (Nov. 1973), pp. 635–658.

[Bal11] Stephanie Balzer. “RUMER: A PROGRAMMING LANGUAGE AND MODULAR VERIFICATION TECHNIQUE BASED ON RELATIONSHIPS”. PhD thesis. Zurich: ETH Zurich, 2011.

[Bat+94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. “The GenVoca Model of Software-System Generators”. In: *IEEE Software* 11.5 (Sept. 1994), pp. 89–94.

[Bäu+97] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. “The Role Object Pattern”. In: *Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP 97).* Pattern Languages of Programs. 1997.

[BBvdT06a] Matteo Baldoni, Guido Boella, and Leendert van der Torre. “powerJava: Ontologically Founded Roles in Object Oriented Programming Languages”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing - SAC ’06.* The 2006 ACM Symposium. Dijon, France: ACM Press, 2006, p. 1414.

[BBvdT06b] Matteo Baldoni, Guido Boella, and Leendert van der Torre. “Roles as a Coordination Construct: Introducing powerJava”. In: *Electronic Notes in Theoretical Computer Science* 150.1 (Mar. 2006), pp. 9–29.

[BC89] Kent Beck and Ward Cunningham. “A Laboratory for Teaching Object-Oriented Thinking”. In: Object-Oriented Programming Systems, Languages and Applications Conference. New Orleans, LA: ACM, 1989, pp. 1–6.

[BD77] Charles W. Bachman and Manilal Daya. “The Role Concept in Data Models”. In: *Proceedings of the Third International Conference on Very Large Data Bases.* International Conference on Very Large Data Bases. Vol. 3. Tokyo, Japan, 1977, pp. 464–476.

[BD95] Daniel Bardou and Christophe Dony. “Propositions Pour Un Nouveau Modèle d’objets Dans Les Langages à Prototypes.” In: *LMO.* 1995, pp. 93–109.

[BD96] Daniel Bardou and Christophe Dony. “Split Objects: A Disciplined Use of Delegation within Objects”. In: *ACM SIGPLAN Notices* 31.10 (Oct. 1, 1996), pp. 122–137.

[BDD00] Johan Brichau, Wolfgang De Meuter, and Kris De Volder. “Jumping Aspects”. In: *Unknown Journal* (2000).

[Bez+12] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. “Julia: A Fast Dynamic Language for Technical Computing”. Sept. 23, 2012. arXiv: `1209.5145 [cs]`.

[BG95] Elisa Bertino and Giovanna Guerrini. “Objects with Multiple Most Specific Classes”. In: *Object-Oriented Programming.* Vol. 952. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 102–126.

[BGE07] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. “A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships”. In: *ECOOP 2007 – Object-Oriented Programming.* Vol. 4609. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 323–346.

[BH02] Jason Baker and Wilson Hsieh. “Runtime Aspect Weaving through Metaprogramming”. In: *Proceedings of the 1st International Conference on Aspect-oriented Software Development.* AOSD ’02. New York, NY, USA: Association for Computing Machinery, Apr. 22, 2002, pp. 86–95.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. *ASM: A Code Manipulation Tool to Implement Adaptable Systems.* France Télécom R&D, DTL/ASR, 2002.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. “JTS: Tools for Implementing Domain-Specific Languages”. In: *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203).* Fifth International Conference on Software Reuse. Victoria, BC, Canada: IEEE Comput. Soc, 1998, pp. 143–153.

[Bob+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. “Common Lisp Object System Specification”. In: *ACM SIGPLAN Notices* 23 (SI Sept. 1988), pp. 1–142.

[Boc+04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. “Virtual Machine Support for Dynamic Join Points”. In: ACM Press, 2004, pp. 83–92.

[Boc+05] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. “Envelope-Based Weaving for Faster Aspect Compilers”. In: *NODe 2005 GSEM 2005*. Net.ObjectDays. Vol. P-69. 2005, pp. 3–18.

[Boc+06] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. “Adapting Virtual Machine Techniques for Seamless Aspect Support”. In: ACM Press, 2006, p. 109.

[Bol+09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. “Tracing the Meta-Level: PyPy’s Tracing JIT Compiler”. In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems - ICOOOLPS ’09*. The 4th Workshop. Genova, Italy: ACM Press, 2009, pp. 18–25.

[Bol+11] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. “Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages”. In: *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems - ICOOOLPS ’11*. The 6th Workshop. Lancaster, United Kingdom: ACM Press, 2011, pp. 1–8.

[Bon04] Jonas Bonér. “AspectWerkz – Dynamic AOP for Java”. In: International Conference on Aspect-Oriented Software Development. 2004.

[Bor86] Alan Borning. “Classes versus Prototypes in Object-Oriented Languages”. In: *Proceedings of the ACM/IEEE Fall Joint Computer Conference*. Fall Joint Computer Conference. Dallas, Texas, Nov. 1986, pp. 36–40.

[BP99] Gianfranco Bilardi and Keshav Pingali. “The Static Single Assignment Form and Its Computation”. In: *Cornell University Technical Report* (1999).

[BPR01] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. “JADE: A FIPA2000 Compliant Agent Development Environment”. In: *Proceedings of the Fifth International Conference on Autonomous Agents*. AGENTS01: Autonomous Agents 2001. Montreal Quebec Canada: ACM, May 28, 2001, pp. 216–217.

[Bri+05] Johan Brichau et al. *Survey of Aspect-oriented Languages and Execution Models.* AOSD-Europe-VUB-01 Deliverable D12. Brussels, Belgium: Vrije Universiteit Brussel, May 17, 2005.

[Bro89] Alfred Leonard Brown. *Persistent Object Stores.* University of St. Andrews (United Kingdom), 1989.

[Bru10] Stefan Brunthaler. "Efficient Interpretation Using Quickening". In: *ACM SIGPLAN Notices* 45.12 (Dec. 3, 2010), p. 1.

[Bur95] M Alison Burkett. "Clarifying Roles and Responsibilities: An Analytical Approach to Clear Any Confusion and Sustain Performance". In: *CMA-ONTARIO-* 69 (1995), pp. 26–26.

[CC95] Cliff Click and Keith D. Cooper. "Combining Analyses, Combining Optimizations". In: *ACM Transactions on Programming Languages and Systems* 17.2 (Mar. 1995), pp. 181–196.

[CGM17] Guido Chari, Diego Garbervetsky, and Stefan Marr. "A Metaobject Protocol for Optimizing Application-Specific Run-Time Variability". In: ACM Press, 2017, pp. 1–5.

[CH05] Pascal Costanza and Robert Hirschfeld. "Language Constructs for Context-Oriented Programming: An Overview of ContextL". In: *Proceedings of the 2005 Conference on Dynamic Languages Symposium - DLS '05.* The 2005 Conference. San Diego, California: ACM Press, 2005, pp. 1–10.

[Cha92] Craig Chambers. "Object-Oriented Multi-Methods in Cecil". In: *ECOOP '92 European Conference on Object-Oriented Programming.* Vol. 615. Berlin/Heidelberg: Springer-Verlag, 1992, pp. 33–56.

[Cha93] Craig Chambers. "Predicate Classes". In: *ECOOP' 93 — Object-Oriented Programming.* Ed. by Oscar M. Nierstrasz. Vol. 707. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 268–296.

[Cha98] Craig Chambers. "Towards Diesel, a next-Generation OO Language after Cecil. Invited Talk". In: *The Fifth Workshop on Foundations of Object-Oriented Languages.* 1998.

[Che+06] Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. "Rava: Designing a Java Extension with Dynamic Object Roles". In: *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems.* Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems. IEEE, 2006, 7 pp.–459.

[CJ02] K. Chandra Sekharaiah and D. Janaki Ram. “Object Schizophrenia Problem in Object Role System Design”. In: *Object-Oriented Information Systems*. Vol. 2425. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 494–506.

[CKC99] Pascal Costanza, Günter Kniesel, and Armin B. Cremers. “Lava Spracherweiterungen Für Delegation in Java”. In: *JIT’99*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 233–242.

[Cli+00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. “MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java”. In: ACM Press, 2000, pp. 130–145.

[CM95] Stefano Ceri and Rainer Manthey. “Chimera: A Model and Language for Active DOOD Systems”. In: *East/West Database Workshop*. Red. by C. J. van Rijsbergen. London: Springer London, 1995, pp. 3–16.

[Cox94] Brad Cox. “No Silver Bullet Revisted”. In: *Scientific American* (1994), p. 86.

[Cox95] Brad J Cox. “There Is a Silver Bullet”. In: *Information technology and society* (1995), pp. 377–386.

[CP95] Cliff Click and Michael Paleczny. “A Simple Graph-Based Intermediate Representation”. In: *ACM SIGPLAN Notices* 30.3 (Mar. 1995), pp. 35–49.

[Cra+97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. “Compiling Java Just in Time”. In: *IEEE Micro* 17.3 (May-June/1997), pp. 36–43.

[CU89] C. Chambers and D. Ungar. “Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language”. In: *ACM SIGPLAN Notices* 24.7 (July 1, 1989), pp. 146–160.

[CUL89] C. Chambers, D. Ungar, and E. Lee. “An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications - OOPSLA ’89*. Conference Proceedings. New Orleans, Louisiana, United States: ACM Press, 1989, pp. 49–70.

[Dam+19] Diego Elias Damasceno Costa, Cor-Paul Bezemer, Philip Leitner, and Artur Andrzejak. “What’s Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks”. In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1.

[De +05] Bruno De Fraine, Wim Vanderperren, Davy Suvée, and Johan Brichau. “Jumping Aspects Revisited”. In: *Dynamic Aspects Workshop.* 2005, pp. 77–86.

[Dij72] Edsger W. Dijkstra. *The Humble Programmer.* EWD340. 1972, p. 15.

[DMB09] Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. “The Art of the Meta-Aspect Protocol”. In: *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development - AOSD ’09.* The 8th ACM International Conference. Charlottesville, Virginia, USA: ACM Press, 2009, p. 51.

[DMB98] Christophe Dony, Jacques Malenfant, and Daniel Bardou. “Classifying Prototype-Based Programming Languages”. In: *Prototype-based Programming: Concepts, Languages and Applications* (1998).

[DN66] Ole-Johan Dahl and Kristen Nygaard. “SIMULA: An ALGOL-based Simulation Language”. In: *Communications of the ACM* 9.9 (Sept. 1966), pp. 671–678.

[DP01] Rowan Davies and Frank Pfenning. “A Modal Analysis of Staged Computation”. In: *Journal of the ACM* 48.3 (May 2001), pp. 555–604.

[DPZ02] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. “A Generic Role Model for Dynamic Objects”. In: *Active Flow and Combustion Control 2018.* Ed. by Rudibert King. Vol. 141. Cham: Springer International Publishing, 2002, pp. 643–658.

[DPZ04] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. “A Role Model and Its Metaclass Implementation”. In: *Information Systems* 29.3 (May 2004), pp. 235–270.

[DR10] Robert Dyer and Hridesh Rajan. “Supporting Dynamic Aspect-Oriented Features”. In: *ACM Transactions on Software Engineering and Methodology* 20.2 (Aug. 2010), pp. 1–34.

[Dub+13a] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. “Graal IR: An Extensible Declarative Intermediate Representation”. In: *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop.* 2013.

[Dub+13b] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. “An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler”. In: ACM Press, 2013, pp. 1–10.

[DWM14] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. “Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler”. In: ACM Press, 2014, pp. 187–193.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. “Predicate Dispatching: A Unified Theory of Dispatch”. In: *ECOOP’98 — Object-Oriented Programming*. Vol. 1445. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 186–211.

[Ern01] Erik Ernst. “Family Polymorphism”. In: *ECOOP 2001 — Object-Oriented Programming*. Vol. 2072. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 303–326.

[Ers82] A.P. Ershov. “Mixed Computation: Potential Applications and Problems for Study”. In: *Theoretical Computer Science* 18.1 (Apr. 1982), pp. 41–67.

[FCK+95] David Ferraiolo, Janet Cugini, D Richard Kuhn, et al. “Role-Based Access Control (RBAC): Features and Motivations”. In: *Proceedings of 11th Annual Computer Security Application Conference*. 1995, pp. 241–48.

[FD99] IR Forman and SH Danforth. *A New Dimension in Object-Oriented Programming Putting Metaclasses to Work*. New York, NY: Addison-Wesley, 1999.

[Flü+17] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. “Correctness of Speculative Optimizations with Dynamic Deoptimization”. In: *Proceedings of the ACM on Programming Languages* 2 (POPL Dec. 27, 2017), pp. 1–28.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language: Everything You Need to Know ; Covers Ruby 1.8 and 1.9*. 1. ed. Beijing Köln: O’Reilly, 2008. 431 pp.

[Fow97] Martin Fowler. “Dealing with Roles”. In: *Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP 97)*. Pattern Languages of Programs. 1997.

[Fut71] Yoshihiko Futamura. “Partial Evaluation of Computation Process-an Approach to a Compiler-Compiler”. In: *Systems, computers, controls* 2.5 (1971), pp. 45–50.

[Fut99] Yoshihiko Futamura. “Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler”. In: *Higher-Order and Symbolic Computation* 12.4 (1999), pp. 381–391.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, 1995. 395 pp.

[GBB98] Giovanna Guerrini, Elisa Bertino, and René Bal. “A Formal Definition of the Chimera Object-Oriented Data Model”. In: *Journal of Intelligent Information Systems* 11.1 (1998), pp. 5–40.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically Rigorous Java Performance Evaluation”. In: *ACM SIGPLAN Notices* 42.10 (Oct. 21, 2007), p. 57.

[GC90] Bracha Gilad and William Cook. “Mixin-Based Inheritance”. In: *Proc. OOPSLA ’90*. OOPSLA. ACM Press, 1990, pp. 303–311.

[GJM03] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Upper Saddle River (N.J.): Prentice Hall, 2003.

[GØ02] Kasper B. Graversen and Kasper Østerbye. “Aspect Modelling as Role Modelling”. In: Tool Support for Aspect Oriented Software Development. 2002.

[GØ03] Kasper B. Graversen and Kasper Østerbye. “Implementation of a Role Language for Object-Specific Dynamic Separation of Concerns”. In: *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*. 2003.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Series in Computer Science. Reading, Mass: Addison-Wesley, 1989. 585 pp.

[Gra03a] Kasper B. Graversen. *Explaining the Implementation of Chameleon — a Short Overview*. Technical Report March 11–2003. Copenhagen, Denmark: IT University of Copenhagen, Mar. 2003.

[Gra03b] Kasper B. Graversen. “The Successes and Failures of a Language as a Language Extension”. In: Workshop on Object-oriented Language Engineering for the Post-Java Era. Darmstadt, Germany, 2003.

[Gra06] Kasper B. Graversen. “The Nature of Roles: A Taxonomic Analysis of Roles as a Language Construct”. PhD thesis. Copenhagen, Denmark: IT University of Copenhagen, 2006.

[Gri+17] Matthias Grimmer, Stefan Marr, Mario Kahlhofer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. “Applying Optimizations for Dynamically-typed Languages to Java”. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes - ManLang 2017*. The 14th International Conference. Prague, Czech Republic: ACM Press, 2017, pp. 12–22.

[GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Röck. “Extending Object-Oriented Systems with Roles”. In: *ACM Transactions on Information Systems* 14.3 (July 1, 1996), pp. 268–296.

[GWB91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. “CLOS: Integrating Object-Oriented and Functional Programming”. In: *Communications of the ACM* 34.9 (Sept. 1, 1991), pp. 29–38.

[Har08] William Harrison. *Subject-Oriented Programming and Design Patterns*. Subject-oriented programming and design patterns. Mar. 16, 2008. URL: `https://web.archive.org/web/20080316055453/http://www.research.ibm.com/sop/sopcpats.htm` (visited on 07/31/2024).

[Hau+05] Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. “An Execution Layer for Aspect-Oriented Programming Languages”. In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. ACM/USENIX International Conference on Virtual Execution Environments. ACM Press, 2005, p. 142.

[Hau05] Michael Haupt. “Virtual Machine Support for Aspect-Oriented Programming Languages”. PhD thesis. Technische Universität Darmstadt, 2005.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. “Context-Oriented Programming.” In: *The Journal of Object Technology* 7.3 (2008), p. 125.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches". In: *ECOOP'91 European Conference on Object-Oriented Programming.* Vol. 512. Berlin/Heidelberg: Springer-Verlag, 1991, pp. 21–38.

[Her03] Stephan Herrmann. "Object Teams: Improving Modularity for Crosscutting Collaborations". In: *Objects, Components, Architectures, Services, and Applications for a Networked World.* Vol. 2591. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 248–264.

[Her04] Stephan Herrmann. *Confinement and Representation Encapsulation in Object Teams.* Technical Report 2004/06. Berlin, Germany: Technische Universität Berlin, Fakultät IV-Elektrotechnik und Informatik, 2004.

[Her05] Stephan Herrmann. "Programming with Roles in ObjectTeams/Java". In: *AAAI Fall Symposium on Roles- an Interdisciplinary Perspective.* 2005.

[Her07] Stephan Herrmann. "A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java". In: *Applied Ontology* 2.2 (2007), pp. 181–207.

[Her10] Stephan Herrmann. "Demystifying Object Schizophrenia". In: *MASPEGHI '10: Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance.* 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance. Maribor, Slovenia: ACM Press, June 22, 2010, pp. 1–5.

[HH04] Erik Hilsdale and Jim Hugunin. "Advice Weaving in AspectJ". In: *Proceedings of the 3rd International Conference on Aspect-oriented Software Development - AOSD '04.* The 3rd International Conference. Lancaster, UK: ACM Press, 2004, pp. 26–35.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems". In: *ACM SIGPLAN Notices* 25.10 (Oct. 1, 1990), pp. 169–180.

[HHM11] Stephan Herrmann, Christine Hundt, and Marco Mosconi. *OT/J Language Definition v1.3.* May 15, 2011.

[Hir01] Robert Hirschfeld. "Aspects-AOP with Squeak". In: *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001).* Citeseer, 2001.

[Hir03] Robert Hirschfeld. "AspectS - Aspect-Oriented Programming with Squeak". In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. Vol. 2591. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 216–232.

[HM00] Stephan Herrmann and Mira Mezini. "PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments". In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA00: Object Oriented Programming Systems Languages and Applications Conference. Minneapolis Minnesota USA: ACM, Oct. 2000, pp. 188–207.

[HM01] Stephan Herrmann and Mira Mezini. "Combining Composition Styles in the Evolvable Language LAC". In: *Proceedings of the 23rd International Conference on Software Engineering*. Workshop on Advanced Separation of Concerns in Software Engineering. Toronto, Canada: IEEE Computer Society, 2001.

[HM04] Michael Haupt and Mira Mezini. "Micro-Measurements for Dynamic Aspect-Oriented Systems". In: *Object-Oriented and Internet-Based Technologies*. Vol. 3263. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 81–96.

[HM05] Michael Haupt and Mira Mezini. "Virtual Machine Support for Aspects with Advice Instance Tables". In: *L'Objet* 11.3 (2005).

[HMI13] Robert Hirschfeld, Hidehiko Masuhara, and Atsushi Igarashi. "L: Context-Oriented Programming with Only Layers". In: *Proceedings of the 5th International Workshop on Context-Oriented Programming - COP'13*. The 5th International Workshop. Montpellier, France: ACM Press, 2013, pp. 1–5.

[HO93] William Harrison and Harold Ossher. "Subject-Oriented Programming: A Critique of Pure Objects". In: *ACM SIGPLAN Notices* 28.10 (Oct. 1, 1993), pp. 411–428.

[Höl93] Urs Hölzle. "Integrating Independently-Developed Components in Object-Oriented Languages". In: *ECOOP' 93 — Object-Oriented Programming*. Ed. by Oscar M. Nierstrasz. Vol. 707. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 36–56.

[HS07] Michael Haupt and Hans Schippers. “A Machine Model for Aspect-Oriented Programming”. In: *ECOOP 2007 – Object-Oriented Programming*. Vol. 4609. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 501–524.

[Hud+92] Paul Hudak et al. “Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language Version 1.2”. In: *ACM SigPlan notices* 27.5 (1992), pp. 1–164.

[Hum+14] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. “A Domain-Specific Language for Building Self-Optimizing AST Interpreters”. In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE’14: Generative Programming: Concepts and Experiences. Västerås Sweden: ACM, Sept. 15, 2014, pp. 123–132.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: A Minimal Core Calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems* 23.3 (May 1, 2001), pp. 396–450.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International Series in Computer Science. New York: Prentice Hall, 1993. 415 pp.

[JGS94] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. 2. [print.] Prentice Hall International Series in Computer Science. New York: Prentice Hall, 1994. 415 pp.

[Jon+16] Timothy Jones, Michael Homer, James Noble, and Kim Bruce. “Object Inheritance Without Classes”. In: *Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany* (2016).

[Jon96] Neil D. Jones. “An Introduction to Partial Evaluation”. In: *ACM Computing Surveys* 28.3 (Sept. 1996), pp. 480–503.

[JT03] Bo Nørregaard Jørgensen and Eddy Truyen. “Evolution of Collective Object Behavior in Presence of Simultaneous Client-Specific Views”. In: *Object-Oriented Information Systems*. Vol. 2817. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 18–32.

[KDB91] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. Cambridge, Mass: MIT Press, 1991. 335 pp.

[Ken99] Elizabeth A. Kendall. “Role Model Designs and Implementations with Aspect-Oriented Programming”. In: ACM Press, 1999, pp. 353–369.

[KG89] Sonya E. Keene and Dan Gerson. *Object-Oriented Programming in Common LISP: A Programmer’s Guide to CLOS*. Reading, Mass: Addison-Wesley, 1989. 266 pp.

[Kic+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. “An Overview of AspectJ”. In: *ECOOP 2001 — Object-Oriented Programming*. Vol. 2072. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 327–354.

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming”. In: *ECOOP’97 — Object-Oriented Programming*. Vol. 1241. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[Kic96] G. Kiczales. “Beyond the Black Box: Open Implementation”. In: *IEEE Software* 13.1 (Jan./1996), pp. 8, 10–11.

[Kla+92] W Klas, EJ Neuhold, R Bahlke, P Fankhauser, G Fischer, M Kaul, P Muth, T Rakow, and V Turau. *VODAK Design Specification Document, VML-VODAK Model Language*. Version 3.0. Working paper GMD, 1992.

[KLH12] Robert Krahn, Jens Lincke, and Robert Hirschfeld. “Efficient Layer Activation in Context JS”. In: IEEE, Jan. 2012, pp. 76–83.

[Kni00] Günter Kniesel. “Dynamic Object-Based Inheritance with Subtyping”. PhD thesis. Bonn, Germany: Universität Bonn, July 7, 2000.

[Kni94] Günter Kniesel. *Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems*. Technical Report IAI-TR-94-3. Bonn, Germany: Universität Bonn, Institut für Informatik III, Oct. 1994.

[Kni96] Günter Kniesel. *Objects Don’t Migrate!* Technical Report IAI-TR-96-11. Bonn, Germany: Universität Bonn, Institut für Informatik III, Apr. 1996.

[Kni99] Günter Kniesel. “Type-Safe Delegation for Run-Time Component Adaptation”. In: *ECOOP’ 99 — Object-Oriented Programming*. Ed. by Rachid Guerraoui. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1628. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 351–366.

[Kör21] Marcel Körner. “Mapping Contextual Role-oriented Programming to Object-based Inheritance in JavaScript”. BA thesis. Dresden, Germany: Technische Universtität Dresden, Mar. 8, 2021.

[KRC91] Günter Kniesel, Mechthild Rohen, and Armin B. Cremers. “A Management System for Distributed Knowledge Base Applications”. In: *Verteilte Künstliche Intelligenz Und Kooperatives Arbeiten*. Vol. 291. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 65–76.

[KT09] Tetsuo Kamina and Tetsuo Tamai. “Towards Safe and Flexible Object Adaptation”. In: International Workshop on Context-Oriented Programming. ACM Press, 2009, pp. 1–6.

[KT10] Tetsuo Kamina and Tetsuo Tamai. “A Smooth Combination of Role-based Language and Context Activation”. In: *FOAL 2010 Proceedings*. 2010.

[Küh+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. “A Metamodel Family for Role-Based Modeling and Programming Languages”. In: *Software Language Engineering*. Vol. 8706. Cham: Springer International Publishing, 2014, pp. 141–160.

[Küh+15] 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*. SLE. Pittsburgh, PA, USA, 2015, pp. 113–124.

[Küh17] Thomas Kühn. “A Family of Role-Based Languages”. PhD thesis. Technische Universtität Dresden, 2017.

[LA15] Max Leuthäuser and Uwe Aßmann. “Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-Based Programming”. In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering. ACM Press, 2015, pp. 25–33.

[Läm02] Ralf Lämmel. “A Semantical Approach to Method-Call Interception”. In: *Proceedings of the 1st International Conference on Aspect-oriented Software Development*. AOSD02: 1st International Conference on Aspect-

Oriented Software Development. Enschede The Netherlands: ACM, Apr. 22, 2002, pp. 41–55.

[Leu17a] Max Leuthäuser. “A Pure Embedding of Roles”. PhD thesis. Dresden: Technische Universtität Dresden, 2017.

[Leu17b] Max Leuthäuser. “Pure Embedding of Evolving Objects”. In: *The Ninth International Conference on Advanced Cognitive Technologies and Applications.* 2017, pp. 22–30.

[Leu18] Max Leuthäuser. *SCROLL Compiler Plugin.* May 3, 2018. URL: `https://github.com/max-leuthaeuser/SCROLLCompilerPlugin` (visited on 09/25/2023).

[Leu19] Max Leuthäuser. *SCROLLGen.* Apr. 15, 2019. URL: `https://github.com/max-leuthaeuser/SCROLLGen` (visited on 01/30/2024).

[Lie86] Henry Lieberman. “Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications - OOPLSA ’86.* Conference Proceedings. Portland, Oregon, United States: ACM Press, 1986, pp. 214–223.

[Lil05] David J Lilja. *Measuring Computer Performance: A Practitioner’s Guide.* Cambridge university press, 2005.

[Lin+11] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. “An Open Implementation for Context-Oriented Layer Composition in ContextJS”. In: *Science of Computer Programming* 76.12 (Dec. 2011), pp. 1194–1209.

[LSU88] Henry Lieberman, Lynn Stein, and David Ungar. “Treaty of Orlando”. In: *ACM SIGPLAN Notices* 23.5 (May 1, 1988), pp. 43–44.

[Mal95] J. Malenfant. “On the Semantic Diversity of Delegation-Based Programming Languages”. In: *ACM SIGPLAN Notices* 30.10 (Oct. 17, 1995), pp. 215–230.

[Mar+22] Stefan Marr, Octave Larose, Sophie Kaleba, and Chris Seaton. “Truffle Interpreter Performance without the Holy Graal”. In: The 2022 Graal Workshop: Science, Art, Magic: Using and Developing The Graal Compiler. Online, Virtual, 2022.

[Mar18] Stefan Marr. “ReBench: Execute and Document Benchmarks Reproducibly”. In: (Aug. 2018).

[MD15] Stefan Marr and Stéphane Ducasse. “Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters”. In: ACM Press, 2015, pp. 821–839.

[MDM16] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. “Cross-Language Compiler Benchmarking: Are We Fast Yet?” In: *ACM SIGPLAN Notices* 52.2 (Nov. 1, 2016), pp. 120–131.

[MHH91] Warwick B. Mugridge, John Hamer, and John G. Hosking. “Multi-Methods in a Statically-Typed Programming Language”. In: *ECOOP’91 European Conference on Object-Oriented Programming.* Ed. by Pierre America. Vol. 512. Berlin/Heidelberg: Springer-Verlag, 1991, pp. 307–324.

[Mil97] Robin Milner. *The Definition of Standard ML: Revised.* MIT press, 1997.

[MK03] Hidehiko Masuhara and Gregor Kiczales. “Modeling Crosscutting in Aspect-Oriented Mechanisms”. In: *ECOOP 2003–Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings 17.* Springer, 2003, pp. 2–28.

[MKD03] H. Masuhara, G. Kiczales, and C. Dutchyn. “A Compilation and Optimization Model for Aspect-Oriented Programs”. In: *Compiler Construction.* Ed. by Görel Hedin. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2622. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 46–60.

[MMI72] David W. Matula, George Marble, and Joel D. Isaacson. “GRAPH COLORING ALGORITHMS”. In: *Graph Theory and Computing.* Elsevier, 1972, pp. 109–122.

[MO02] Mira Mezini and Klaus Ostermann. “Integrating Independent Components with On-Demand Remodularization”. In: *ACM SIGPLAN Notices* 37.11 (Nov. 17, 2002), p. 52.

[MO03] Mira Mezini and Klaus Ostermann. “Conquering Aspects with Caesar”. In: *Proceedings of the 2nd International Conference on Aspect-oriented Software Development - AOSD ’03.* The 2nd International Conference. Boston, Massachusetts: ACM Press, 2003, pp. 90–99.

[MO92] J Morton and JJ Odell. *Object Oriented Analysis and Design.* Vol. 1140. Englewood Cliffs (New Jersey): Prentice-Hall, 1992.

[Moo+05] Adriaan Moors, Jan Smans, Eddy Truyen, Frank Piessens, and Wouter Joosen. "Safe Language Support for Feature Composition through Feature-Based Dispatch". In: 2nd Workshop on Managing Variabilities Consistently in Design and Code, OOPSLA. San Diego, CA, USA, 2005.

[Moo86] David A. Moon. "Object-Oriented Programming with Flavors". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications - OOPLSA '86*. Conference Proceedings. Portland, Oregon, United States: ACM Press, 1986, pp. 1–8.

[Moz24] Mozilla. *JavaScript Reference*. 2024. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference` (visited on 07/02/2024).

[MSD15] Stefan Marr, Chris Seaton, and Stéphane Ducasse. "Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises". In: ACM Press, 2015, pp. 545–554.

[MSL01] M Mezini, L Seiter, and K Lieberherr. "Software Architecture and Component Technology: State of the Art in Research and Practice, Chapter Component Integration with Pluggable Composite Adapters". In: *Software Architecture and Component Technology: State of the Art in Research and Practice. Kluwer Academic Publishers* (2001).

[MT08] Supasit Monpratarnchai and Tamai Tetsuo. "The Implementation and Execution Framework of a Role Model Based Language, EpsilonJ". In: IEEE, 2008, pp. 269–276.

[Mus+08] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. "Multiple Dispatch in Practice". In: *ACM SIGPLAN Notices* 43.10 (Oct. 27, 2008), p. 563.

[MY98] Hidehiko Masuhara and Akinori Yonezawa. "Design and Partial Evaluation of Meta-Objects for a Concurrent Reflective Language". In: *ECOOP'98 — Object-Oriented Programming*. Vol. 1445. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 418–439.

[NA05] Angela Nicoară and Gustavo Alonso. "Dynamic AOP with PROSE". In: *Proc. of 1st International Workshop on Adaptive and Self-Managing Enterprise Applications*. Workshop on Adaptive and Self-Managing Enterprise Applications. Porto, Portugal, 2005.

[New21] Chris Newland. *The Best HotSpot JVM Options and Switches for Java 11 through Java 17*. Aug. 27, 2021. URL: `https://blogs.oracle.com/javamagazine/post/the-best-hotspot-jvm-options-and-switches-for-java-11-through-java-17` (visited on 09/13/2023).

[Ora23a] Oracle. *Java Microbenchmark Harness (JMH)*. 2023. URL: `https://github.com/openjdk/jmh` (visited on 09/14/2023).

[Ora23b] Oracle. *Java on Truffle*. Oracle Help Center. 2023. URL: `https://docs.oracle.com/en/graalvm/jdk/21/docs/reference-manual/java-on-truffle/` (visited on 06/05/2024).

[Ora23c] Oracle. *VisualVM All-in-One Java Troubleshooting Tool*. Sept. 19, 2023. URL: `https://visualvm.github.io` (visited on 09/26/2023).

[OT99] Harold Ossher and Peri Tarr. *Multi-Dimensional Separation of Concerns in Hyperspace*. Research Report RC 21452(96717)16APR99. New York, NY, USA: IBM T.J. Watson Research Center, 1999.

[Pae93] Andreas Paepcke, ed. *Object-Oriented Programming: The CLOS Perspective*. Cambridge, Mass: MIT Press, 1993. 352 pp.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. “Just-in-Time Aspects: Efficient Dynamic Weaving for Java”. In: ACM Press, 2003, pp. 100–109.

[Pap91] M. P. Papazoglou. “Roles: A Methodology for Representing Multifaceted Objects”. In: *Database and Expert Systems Applications*. Ed. by Dimitris Karagiannis. Vienna: Springer Vienna, 1991, pp. 7–12.

[Paw+01] Renaud Pawlak, Laurence Duchien, Gérard Florin, and Lionel Seinturier. “JAC: A Flexible Solution for Aspect-Oriented Programming in Java”. In: *Metalevel Architectures and Separation of Crosscutting Concerns*. Ed. by Akinori Yonezawa and Satoshi Matsuoka. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2192. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–24.

[Per90] B. Pernici. “Objects with Roles”. In: *Proceedings of the Conference on Office Information Systems -*. The Conference. Cambridge, Massachusetts, United States: ACM Press, 1990, pp. 205–215.

[PFH16] Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. “Optimizing Sideways Composition: Fast Context-oriented Programming in ContextPyPy”. In: ACM Press, 2016, pp. 13–20.

[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. “Dynamic Weaving for Aspect-Oriented Programming”. In: *Proceedings of the 1st International Conference on Aspect-oriented Software Development - AOSD ’02*. The 1st International Conference. Enschede, The Netherlands: ACM Press, 2002, p. 141.

[Pie+12] Christian Piechnick, Sebastian Richly, Sebastian Götz, Claas Wilke, and Uwe Aßmann. “Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation-Smart Application Grids”. In: *Proceedings of ADAPTIVE* (2012), pp. 93–102.

[Piv19] Software Inc. Pivotal. *Aspect Oriented Programming with Spring*. Spring Framework Documentation. 2019. URL: `https://docs.spring.io/spring-framework/reference/core/aop.html` (visited on 02/18/2019).

[PO08] Michael Pradel and Martin Odersky. “SCALA ROLES A Lightweight Approach towards Reusable Collaborations”. In: *ICSOFT 2008 - Proceedings of the 3rd International Conference on Software and Data Technologies*. 2008, pp. 13–20.

[Pra08] Michael Pradel. “Roles and Collaborations in Scala”. Diplom Thesis. Dresden: Technische Universtität Dresden, 2008.

[PS99] Massimiliano Poletto and Vivek Sarkar. “Linear Scan Register Allocation”. In: *ACM Transactions on Programming Languages and Systems* 21.5 (Sept. 1999), pp. 895–913.

[PSS07] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. “Open Multi-Methods for C++”. In: ACM Press, 2007, p. 123.

[R C18] R Core Team. *R: A Language and Environment for Statistical Computing*. manual. Vienna, Austria: R Foundation for Statistical Computing, 2018.

[Raf23] Winterhalter Rafael. *Runtime Code Generation for the Java Virtual Machine*. ByteBuddy. 2023.

[RG98] Dirk Riehle and Thomas Gross. “Role Model Based Framework Design and Integration”. In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press, 1998, pp. 117–133.

[RO13] Jose Manuel Redondo and Francisco Ortin. “Efficient Support of Dynamic Inheritance for Class- and Prototype-Based Languages”. In: *Journal of Systems and Software* 86.2 (Feb. 2013), pp. 278–301.

[Ros09a] John Rose. *The Da Vinci Machine Project.* 2009. URL: `https://openjdk.org/projects/mlvm/` (visited on 11/12/2023).

[Ros09b] John R. Rose. "Bytecodes Meet Combinators: Invokedynamic on the JVM". In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages.* Third Workshop on Virtual Machines and Intermediate Languages. Orlando, Florida: ACM Press, 2009, pp. 1–11.

[Ros19] John R. Rose. *JEP 243: Java-Level JVM Compiler Interface.* 2019. URL: `https://openjdk.org/jeps/243` (visited on 05/24/2024).

[RWL96] Trygve Reenskaug, Per Wold, and Odd Arilc Lehne. *Working with Objects: The OOram Software Engineering Method.* Greenwich: Manning, 1996. 366 pp.

[SB02] Yannis Smaragdakis and Don Batory. "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs". In: *ACM Transactions on Software Engineering and Methodology* 11.2 (Apr. 1, 2002), pp. 215–255.

[SC17] Lars Schütze and Jeronimo Castrillon. "Analyzing State-of-the-Art Role-based Programming Languages". In: *Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming '17.* The International Conference. Brussels, Belgium: ACM Press, 2017, pp. 1–6.

[SC19] Lars Schütze and Jeronimo Castrillon. "Efficient Late Binding of Dynamic Function Compositions". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2019.* The 12th ACM SIGPLAN International Conference. Athens, Greece: ACM Press, 2019, pp. 141–151.

[SC20] Lars Schütze and Jeronimo Castrillon. "Efficient Dispatch of Multi-object Polymorphic Call Sites in Contextual Role-Oriented Programming Languages". In: *17th International Conference on Managed Programming Languages and Runtimes.* MPLR '20: 17th International Conference on Managed Programming Languages and Runtimes. Virtual UK: ACM, Nov. 4, 2020, pp. 52–62.

[SC23] Lars Schütze and Jeronimo Castrillon. "Towards Virtual Machine Support for Contextual Role-Oriented Programming Languages". In: *Proceedings of the 15th ACM International Workshop on Context-Oriented Programming and Advanced Modularity (COP '23).* ACM International

Workshop on Context-Oriented Programming and Advanced Modularity. Seattle, WA, USA: ACM, July 17, 2023.

[SE04] Yunus Emre Selçuk and Nadia Erdoğan. “JAWIRO: Enhancing Java with Roles”. In: *Computer and Information Sciences - ISCIS 2004*. Vol. 3280. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 927–934.

[SE06] Yunus Emre Selçuk and Nadia Erdoğan. “A Role Model for Description of Agent Behavior and Coordination”. In: *Engineering Societies in the Agents World VI*. Ed. by Oğuz Dikenelli, Marie-Pierre Gleizes, and Alessandro Ricci. Red. by David Hutchison et al. Vol. 3963. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 29–48.

[Sea16] Chris Seaton. “AST Specialisation and Partial Evaluation for Easy High-Performance Metaprogramming”. In: Workshop on Meta-Programming Techniques and Reflection. Amsterdam, Netherlands, 2016.

[SG93] Guy L. Steele and Richard P. Gabriel. “The Evolution of Lisp”. In: *ACM SIGPLAN Notices* 28.3 (1993), pp. 231–270.

[SGP12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. “Context-Oriented Programming: A Software Engineering Perspective”. In: *Journal of Systems and Software* 85.8 (Aug. 2012), pp. 1801–1817.

[Sha96] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison Wesley Longman Publishing Co., Inc., 1996.

[SKC22] Lars Schütze, Cornelius Kummer, and Jeronimo Castrillon. “Guard the Cache: Dispatch Optimization in a Contextual Role-oriented Language”. In: *COP 2022: International Workshop on Context-Oriented Programming and Advanced Modularity (Collocated with ECOOP)*. COP ’22: International Workshop on Context-Oriented Programming and Advanced Modularity. Berlin Germany: ACM, June 7, 2022, pp. 27–34.

[SO04] Juan Manuel Serrano and Sascha Ossowski. “On the Impact of Agent Communication Languages on the Implementation of Agent Systems”. In: *Cooperative Information Agents VIII*. Vol. 3191. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 92–106.

[SS98] Gerald Jay Sussman and Guy L. Steele Jr. “Scheme: A Interpreter for Extended Lambda Calculus”. In: *Higher Order Symbolic Computation* 11.4 (1998), pp. 405–439.

[ST04] Michael Schrefl and Thomas Thalhammer. “Using Roles in Java”. In: *Software: Practice and Experience* 34.5 (Apr. 25, 2004), pp. 449–464.

[ST95] Michael Sperber and Peter Thiemann. “The Essence of LR Parsing”. In: *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation - PEPM ’95*. The 1995 ACM SIGPLAN Symposium. La Jolla, California, United States: ACM Press, 1995, pp. 146–155.

[Ste00a] Friedrich Steimann. “Formale Modellierung Mit Rollen”. Habilitation. Hannover: Universität Hannover, 2000.

[Ste00b] Friedrich Steimann. “On the Representation of Roles in Object-Oriented and Conceptual Modelling”. In: *Data & Knowledge Engineering* 35.1 (Oct. 2000), pp. 83–106.

[Ste01] Friedrich Steimann. “Role= Interface: A Merger of Concepts”. In: *Journal of Object-Oriented Programming* (2001).

[Ste87] Lynn Andrea Stein. “Delegation Is Inheritance”. In: *ACM SIGPLAN Notices* 22.12 (Dec. 1987), pp. 138–146.

[SU96] Randall B Smith and David Ungar. “A Simple and Unifying Approach to Subjective Objects”. In: *TAPOS* 2.3 (1996), pp. 161–178.

[SVL08] R.S. Sangwan, P. Vercellone-Smith, and P.A. Laplante. “Structural Epochs in the Complexity of Software over Time”. In: *IEEE Software* 25.4 (July 2008), pp. 66–73.

[SWM14] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. “Partial Escape Analysis and Scalar Replacement for Java”. In: *Proceedings of the 12th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2014*. CGO. 2014.

[Tai+16a] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. “A Dynamic Instance Binding Mechanism Supporting Run-Time Variability of Role-Based Software Systems”. In: *Companion Proceedings of the 15th International Conference on Modularity*. ACM Press, 2016, pp. 137–142.

[Tai+16b] Nguonly Taing, Markus Wutzler, Thomas Springer, Nicolás Cardozo, and Alexander Schill. “Consistent Unanticipated Adaptation for Context-Dependent Applications”. In: *Proceedings of the 8th International Workshop on Context-Oriented Programming*. ECOOP ’16: European Conference on Object-Oriented Programming. Rome Italy: ACM, July 17, 2016, pp. 33–38.

[Tai+17] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. "A Rollback Mechanism to Recover from Software Failures in Role-based Adaptive Software Systems". In: *Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming '17*. The International Conference. Brussels, Belgium: ACM Press, 2017, pp. 1–6.

[Tai17] Nguonly Taing. "Run-Time Variability with Roles". PhD thesis. Dresden: Technische Universtität Dresden, 2017.

[Tar+99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. International Conference on Software Engineering. Los Angeles, USA: IEEE, May 1999, pp. 107–119.

[Tat+00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. "OpenJava: A Class-Based Macro System for Java". In: *Reflection and Software Engineering*. Ed. by Walter Cazzola, Robert J. Stroud, and Francesco Tisato. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1826. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 117–133.

[TJJ00] E. Truyen, B.N. Jorgensen, and W. Joosen. "Customization of Component-based Object Request Brokers through Dynamic Reconfiguration". In: *Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33*. 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33. Mont-Saint-Michel, France: IEEE Comput. Soc, 2000, pp. 181–194.

[Tor21] Mads Torgersen. *Proposal: Roles and Extensions*. [Proposal]: Roles and extensions. Dec. 1, 2021. URL: `https://github.com/dotnet/csharplang/discussions/5496` (visited on 08/22/2023).

[TR10] Christian Thalinger and John Rose. "Optimizing Invokedynamic". In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java - PPPJ '10*. PPPJ. Vienna, Austria: ACM Press, 2010, p. 1.

[Tru+01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B.N. Jorgensen. “Dynamic and Selective Combination of Extensions in Component-Based Applications”. In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* 23rd International Conference on Software Engineering. ICSE 2001. Toronto, Ont., Canada: IEEE Comput. Soc, 2001, pp. 233–242.

[TUI05] T. Tamai, N. Ubayashi, and R. Ichiyama. “An Adaptive Object Model with Dynamic Role Binding”. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 27th International Conference on Software Engineering, 2005. ICSE 2005. St. Louis, MO, USA: IEEe, 2005, pp. 166–175.

[TUI07] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. “Objects as Actors Assuming Roles in the Environment”. In: *Software Engineering for Multi-Agent Systems V.* Vol. 4408. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 185–203.

[UOK14] 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.* ACM Press, 2014, pp. 113–131.

[US87] David Ungar and Randall B. Smith. “Self: The Power of Simplicity”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications.* OOPSLA87: Object-Oriented Programming Systems, Languages and Applications. Orlando Florida USA: ACM, Dec. 1987, pp. 227–242.

[Van+07] Guido Van Rossum et al. “Python Programming Language.” In: *USENIX Annual Technical Conference.* Vol. 41. 1. Santa Clara, CA, 2007, pp. 1–36.

[Vas04] Alexandre Vasseur. “Dynamic AOP and Runtime Weaving for Java - How Does AspectWerkz Address It?” In: *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development.* Aspect-Oriented Software Development. Lancaster UK, Mar. 2004.

[VDD04] Ellen Van Paesschen, Wolfgang De Meuter, and Theo D’Hondt. “Domain Modeling in Self Yields Warped Hierarchies”. In: *MASPEGHI 2004 Mechanisms for Speialization, Generalization and Inheritance.* Mechanisms for Speialization, Generalization and Inheritance. Berlin, Heidelberg, 2004.

[Vei14] Matthias Veit. *Homepage of RubyObjectTeams.* 2014. URL: `https://sourceforge.net/projects/robjectteam/`.

[Vil+09] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. "Advanced Runtime Adaptation for Java". In: *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering.* GPCE '09. New York, NY, USA: Association for Computing Machinery, Oct. 4, 2009, pp. 85–94.

[VN95] Michael VanHilst and David Notkin. *Using C ++ Templates to Implement Role-Based Designs.* Technical Report 95-07-02. University of Washington: Department of Computer Science & Engineering, 1995.

[VN96a] Michael VanHilst and David Notkin. "Decoupling Change from Design". In: *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering - SIGSOFT '96.* The 4th ACM SIGSOFT Symposium. San Francisco, California, United States: ACM Press, 1996, pp. 58–69.

[VN96b] Michael VanHilst and David Notkin. "Using C++ Templates to Implement Role-Based Designs". In: *Object Technologies for Advanced Software.* Ed. by Kokichi Futatsugi and Satoshi Matsuoka. Red. by Gerhard Goos, Juris Hartmanis, and Jan Leeuwen. Vol. 1049. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 22–37.

[VN96c] Michael VanHilst and David Notkin. "Using Role Components in Implement Collaboration-Based Designs". In: *ACM SIGPLAN Notices* 31.10 (Oct. 1, 1996), pp. 359–369.

[WC98] R.K. Wong and H.L. Chau. "Method Dispatching and Type Safety for Objects with Multiple Roles". In: *Proceedings. Technology of Object-Oriented Languages and Systems, TOOLS 25 (Cat. No.97TB100239).* Technology of Object-Oriented Languages and Systems TOOLS 25. Melbourne, Vic., Australia: IEEE Comput. Soc, 1998, pp. 286–296.

[WCL96] R.K. Wong, H.L. Chau, and F.H. Lochovsky. *DOOR: A Dynamic Object-Oriented Data Model with Roles.* Technical Report HKUST-CS96-12. Hong Kong: Department of Computer Science Hong Kong University of Science and Technology, May 8, 1996.

[WCL97a] R.K. Wong, H.L. Chau, and F.H. Lochovsky. “A Data Model and Semantics of Objects with Dynamic Roles”. In: *Proceedings 13th International Conference on Data Engineering.* 13th International Conference on Data Engineering. Birmingham, UK: IEEE Comput. Soc. Press, 1997, pp. 402–411.

[WCL97b] R.K. Wong, H.L. Chau, and F.H. Lochovsky. “Dynamic Knowledge Representation in DOOR”. In: *Proceedings 1997 IEEE Knowledge and Data Engineering Exchange Workshop.* 1997 IEEE Knowledge and Data Engineering Exchange Workshop. Newport Beach, CA, USA: IEEE Comput. Soc, 1997, pp. 89–96.

[Weg87] Peter Wegner. “Dimensions of Object-Based Language Design”. In: *ACM SIGPLAN Notices* 22.12 (Dec. 1987), pp. 168–182.

[WF10] Christian Wimmer and Michael Franz. “Linear Scan Register Allocation on SSA Form”. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* CGO ’10: 8th Annual IEEE/ ACM International Symposium on Code Generation and Optimization. Toronto Ontario Canada: ACM, Apr. 24, 2010, pp. 170–179.

[Wim+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. “Maxine: An Approachable Virtual Machine for, and in, Java”. In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 1, 2013), pp. 1–24.

[Wöß+14] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. “An Object Storage Model for the Truffle Language Implementation Framework”. In: ACM Press, 2014, pp. 133–144.

[WP17] Brandon M. Williams and Saverio Perugini. “Revisiting the Futamura Projections: A Diagrammatic Approach”. In: *Theoretical and Applied Informatics* 28.4 (Dec. 12, 2017), pp. 15–32. arXiv: `1611.09906`.

[Wür+10] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. “Improving Aspect-Oriented Programming with Dynamic Code Evolution in an Enhanced Java Virtual Machine”. In: *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution - RAM-SE ’10.* The 7th Workshop. Maribor, Slovenia: ACM Press, 2010, pp. 1–5.

,

[Wür+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! '13*. The 2013 ACM International Symposium. Indianapolis, Indiana, USA: ACM Press, 2013, pp. 187–204.

[Wür+17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. “Practical Partial Evaluation for High-Performance Dynamic Language Runtimes”. In: ACM Press, 2017, pp. 662–676.

[Wür14] Thomas Würthinger. “Graal and Truffle: Modularity and Separation of Concerns as Cornerstones for Building a Multipurpose Runtime”. In: ACM Press, 2014, pp. 3–4.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. “Dynamic Code Evolution for Java”. In: ACM Press, 2010, p. 10.

[XBH16] Shijie Xu, David Bremner, and Daniel Heidinga. “MHDeS: Deduplicating Method Handle Graphs for Efficient Dynamic JVM Language Implementations”. In: ACM Press, 2016, pp. 1–10.

[ZO04] Matthias Zenger and Martin Odersky. *Independently Extensible Solutions to the Expression Problem*. 2004.