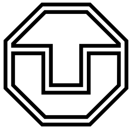

TECHNISCHE UNIVERSITÄT DRESDEN

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

EXCERPT

A PURE EMBEDDING OF ROLES

EXPLORING 4-DIMENSIONAL DISPATCH FOR ROLES IN STRUCTURED CONTEXTS

Max Leuthäuser

Born on: 20th April 1989 in Dresden

DISSERTATION

to achieve the academic degree

DOKTOR-INGENIEUR (DR.-ING.)

Referee

Prof. Anthony Sloane

Supervising professors

Prof. Uwe Aßmann
Prof. Christel Baier

Submitted on: 31st May 2017

ABSTRACT

Present-day software systems have to fulfill an increasing number of requirements, which makes them more and more complex. Many systems need to anticipate changing contexts or need to adapt to changing business rules or requirements. The challenge of 21th-century software development will be to cope with these aspects. We believe that the role concept offers a simple way to adapt an object-oriented program to its changing context. In a role-based application, an object plays multiple roles during its lifetime. If the contexts are represented as first-class entities, they provide dynamic views to the object-oriented program, and if a context changes, the dynamic views can be switched easily, and the software system adapts automatically. However, the concepts of roles and dynamic contexts have been discussed for a long time in many areas of computer science. So far, their employment in an existing object-oriented language requires a specific runtime environment. Also, classical object-oriented languages and their runtime systems are not able to cope with essential role-specific features, such as true delegation or dynamic binding of roles. In addition to that, contexts and views seem to be important in software development. The traditional code-oriented approach to software engineering becomes less and less satisfactory. The support for multiple views of a software system scales much better to the needs of today's systems. However, it relies on programming languages to provide roles for the construction of views. As a solution, this thesis presents an implementation pattern for role-playing objects that does not require a specific runtime system, the *SCala ROles Language* (*SCROLL*). Via this library approach, roles are embedded in a statically typed base language as dynamically evolving objects. The approach is pure in the sense that there is no need for an additional compiler or tooling. The implementation pattern is demonstrated on the basis of the Scala language. As technical support from Scala, the pattern requires dynamic mixins, compiler-translated function calls, and implicit conversions. The details how roles are implemented are hidden in a Scala library and therefore transparent to *SCROLL* programmers. The *SCROLL* library supports roles embedded in structured contexts. Additionally, a four-dimensional, context-aware dispatch at runtime is presented. It overcomes the subtle ambiguities introduced with the rich semantics of role-playing objects. *SCROLL* is written in Scala, which blends a modern object-oriented with a functional programming language. The size of the library is below 1400 lines of code so that it can be considered to have minimalistic design and to be easy to maintain. Our approach solves several practical problems arising in the area of dynamical extensibility and adaptation.

1 THESIS TOPIC AND CONTRIBUTIONS

In the modern software world, software systems are required to adapt to a changing environment. During the lifetime of a software system, new features are requested, existing requirements change, as well as the underlying hardware and operating systems are regularly being renewed. Software and software libraries once written for a specific purpose may become useful in situations, the developer did not anticipate at the time of their creation. One programming paradigm, object-oriented programming, is widely being used to build extensible and flexible software systems. It was and still is successful, because it supports programming with data structures that closely resemble the problem domain. However, future software systems require a higher level of dynamism, which is not offered by classic object-oriented concepts. Dynamically typed, object-oriented scripting languages, such as Ruby and Python, have gained popularity not only because of their ease of use, and have created vibrant communities. They enable the extension of modules, classes, and object through concepts such as duck-typing [43]. But programming in a dynamically typed language comes at a cost: without static type information, it is not possible to analyze programs statically and catch many classes of programming errors (e.g., type errors) early on. The burden is solely left to the programmer. The influence of roles on language design is in the focus of this

work, especially the accompanying problems and ambiguities when dealing with dynamically evolving objects. Role-based programming has been proposed as an extension to object-oriented programming, introducing extensionality in a controlled and well-defined manner. It has been motivated by an easy-to-understand analogy. Also in the real world, objects play different roles in different contexts. In essence, it enables objects to modify and extend their behaviors dynamically at runtime, without the limits imposed by the class hierarchy. How to simply represent roles in existing language runtime environments remains as an open question. Current implementations rely on proxies, reflection, runtime weaving and runtime code generation to support mechanisms, such as true delegation and dynamic role dispatch. This requires additional management code and leads to more problems, such as incomprehensible error messages with polluted stack traces. Furthermore, existing role-based programming languages only support a small subset of the desired role features. Especially, they lack a well-defined concept for context- and role-aware method dispatch to overcome ambiguities, which are introduced with roles. These points are the major roadblock for the wider adoption of role-based programming. The goal of this thesis is to research how roles can be represented at runtime and supported by a rich dispatching concept. A prototypical implementation, called *SCROLL*[1], was developed and is introduced by various examples and an in-depth evaluation. This thesis provides the following main contributions:

SCROLL and the _SCROLL_ MOP First, the embedded method-call interception Domain-Specific Language (DSL) *SCROLL* and its underlying Metaobject-Protocol (MOP) [33, 36, 26] are presented. They are implemented in a lightweight library that allows for pure embedding [23] of roles in a modern, statically typed, object-oriented language (Scala). Only features that are available through Scala's standard compiler are utilized. *SCROLL* allows for easy integration of legacy code and provides a high degree of separation of concerns.

A coupling of static and dynamic role typing With the specification of context-dependent behavior and structure in separate role types, static type checking and program analysis is limited. Coupled static and dynamic role typing supports the developer with the best of both worlds. He benefits from the aforementioned advantages of a statically-typed host language, while at the same time, he profits from the flexibility of dynamic objects.

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

A role-based dispatch at runtime A declarative and parameterizable four-dimensional dispatch for roles in structured contexts is described. This approach relies on the representation of dispatch rules as function objects [49].

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

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

[1] `https://github.com/max-leuthaeuser/SCROLL`

2 BACKGROUND AND PROBLEM ANALYSIS

Dynamic and adaptive infrastructures are the cornerstone of today's software development, e.g., in the Web 3.0 - the internet of things. In most classic class-based object-oriented systems, the association between instances of a class and the class itself is permanent [16]. Such systems hardly cope with new requirements during runtime. Class hierarchies need to be carefully planned and laid-out for dynamic extensions. Indeed, they grow exponentially in case the objects they describe are changing [13]. Two main principles of abstraction are known to structure object-oriented programs: *classification* and *generalization*. The first describes the principle to group objects to classes sharing behavior and attributes. The second principle means the organization of those classes into class hierarchies with grouping the common behavior and attributes into a new superclass, which is then shared among all subclasses. However, when describing real-world objects embedded in a fast and frequently changing environment as well as their classification in a class hierarchy, the permanent association between instances and their classes appears to be too inflexible, because it cannot cope with the requirements of those fast growing or changing systems. When extending an object, the object must be replaced. Then, the internal state of objects needs to be copied, which renders their management cumbersome and error-prone. It becomes even more difficult if there are several of those changes at the same time for the same object. Introducing separate classes for each combination of new behavior for adaptation leads to fast growing class hierarchies, which is undesired.

2.1 FOUNDATIONS OF ROLES

The concept of roles was introduced by [2] as an extension to the network data model. It enables the addition and removal of behavior and attributes at runtime to objects providing a substantial advantage over traditional programming languages, such as Java. Over the recent decades, a lot of role-based approaches have been proposed in the literature, all providing a different notion of roles. As a sophisticated role notion as base for this thesis, the Compartment Role Object Model (CROM) [29] is discussed.

Each object-oriented software design has to solve the static and dynamic aspects of the following problems [44]. The interactions of a class or its instances with the client are non-trivial (*class complexity* **/P.1/**). Each of those clients handle the instances differently and with different use-cases in mind. This interaction needs to be described and constrained from the viewpoint of all relevant contexts. Furthermore, a sound understanding of *object collaborations* and relations at runtime is crucial (**/P.2/**). Adaptive software systems need to be able to define and check those collaborations. Once they are understood, it is important to group and separate them for the sake of comprehensibility (*separation of concerns* **/P.3/**). Relations need to be assigned to relevant contexts. Derived from the two aforementioned tasks, *reuse* is one of the most important attributes, on which adaptivity of the resulting system benefits from (**/P.4/**). With the *invariants and constraints*, behavior of collaborating objects can be constrained and checked during runtime (**/P.5/**). Finally, in adaptive software systems, it is impossible to foresee each and every possible future use-case and context. Hence, all problems, listed so far, define basic requirements that allow for adapting to new and unforeseen contexts (**/P.6/**).

The following describes the motivation for roles in general and introduces their major ingredients in more detail. Abstracting and simplifying the representation of the real-world for some given use-case is, in general, called *modeling*. A model omits irrelevant parts of the captured world. An *object-oriented model* abstracts objects to classes and types. Hence, the complex relationships of real-world objects are simplified to the pertinent needs for the desired purpose. The result of the modeling process is then used during the analysis or design phase in software development. Here,

two major goals can be targeted. First, capturing real situations of a domain for communication during the analysis and, secondly, representing the design of the software system itself, describing its elements and interactions [48]. Traditional modeling languages, such as the Unified Modeling Language (UML) or Entity-Relationship Model (ER), only consider entities of fixed types (static typing), pre-defined structure (attributes), and behavior (methods) without being able to change them dynamically. Every aspect of an entity needs to be integrated at design time, even though some attributes or functions are not needed in certain contexts. Often, the adaption is handled by static inheritance. This is problematic since, e.g., subtyping leads to an enormous amount of subtypes for each and every new context the entity is intended to interact in (*combinatorial explosion of subtypes*). Extensibility becomes, consequently, impracticable. Finally, one might not be able to decide whether a specialization or generalization is applicable. Using design patterns can solve certain problems, but introduces additional overhead with regard to maintenance and readability. This underlines the need for more flexibility and dynamics to improve modeling.

In classic programming languages, the situation is quite similar. During its lifetime, an object might need to change its behavior and attributes for a certain period. In addition, entities of the same type may have different attributes and behavior at the same time. Those aspects are usually not directly supported within classical object-oriented programming languages with static typing. An entity is part of a certain type throughout its lifetime. When recreating it with a different type at runtime, it loses its identity, state needs to be copied, and clients have to handle the newly created identity causing the need for additional management code. An additional problem occurs when an object is conceptually separated into multiple individual sub-objects (*split-object problem* [8]). This requires even more additional management code complicating code maintenance. The split-object problem is very much similar to the issue of *object schizophrenia*: *"Object schizophrenia results when the state and/or behavior of what is intended to appear as a single object are actually broken into several objects (each of which has its own object identity)."* [18].

In the end, with modern software systems becoming more and more complex and having to adapt to continuously changing environments, the resulting problems during design- and runtime cannot be managed easily anymore. Approaches, such as dynamic aspect-orientation [41] and context-oriented programming [22] have been introduced by researchers in the past to handle the aforementioned problems of too static software systems. As an alternative, role-oriented programming [47, 48, 7, 17, 16, 40] can be employed, which is in the focus of this thesis. With the idea of separation of concerns in mind, dynamic and flexible parts of an object are modeled separately from the entity's core. Several parts of an entity have different lifetimes and may only exist during a certain period of the entity's lifetime. These entities are split into natural types (entity's core type) and role types. This enables context-dependent structural and behavioral adaptation. Thus, the dynamic evolution of entities over time becomes an integral part of modeling and programming. Those evolving objects are explicitly modeled and represented at runtime accordingly. The role as a concept, modeling primitive, and first-class citizen in programming captures the context-dependence and dynamic parts of objects with their specific behavior and structure in separate types. Hence, role-based type systems explicitly model behavioral adaptation, in contrast to traditional static type-systems. Entities can evolve during runtime without changing their natural type at all. Instead, they start and stop playing roles. This enables modeling and implementing complex, context-dependent behavior of objects in frequently changing environments. In summary, the role concept is essentially an extension to object orientation enabling objects to adapt their behaviors dynamically. At runtime, roles are bound to objects which then become role-players. A role is defined completely independent of its player and gets filled with life by the player adopting its behavior and structure. We use roles to dynamically add and remove behavior and structure during the runtime and lifetime of an object.

Each approach on role-oriented programming that appeared in the literature during the last decades utilizes its own interpretation of roles leading to an inhomogeneous research landscape. Hence, there is no common notion of what a role is. For instance, ER or UML only consider roles as

Figure 2.1: Steimann's classifying features (1-15) [48], additional ones (16-26) with regard to the context-dependent nature of roles [29], and those role features solely focusing on runtime aspects (27-54) [17].

1. Roles have properties and behaviors.
2. Roles depend on relationships.
3. Objects may play different roles simultaneously.
4. Objects may play the same role (type) several times.
5. Objects may acquire and abandon roles dynamically.
6. The sequence of role acquisition/removal may be restricted.
7. Unrelated objects can play the same role.
8. Roles can play roles.
9. Roles can be transferred between objects.
10. The state of an object can be role-specific.
11. Features of an object can be role-specific.
12. Roles restrict access.
13. Different roles may share structure and behavior.
14. An object and its roles share identity.
15. An object and its roles have different identities.

16. Relationships between roles can be constrained.
17. There may be constraints between relationships.
18. Roles can be grouped and constrained together.
19. Roles depend on compartments.
20. Compartments have properties and behaviors.
21. A role can be part of several compartments.
22. Compartments may play roles like objects.
23. Compartments may play roles which are part of themselves.
24. Compartments can contain other compartments.
25. Different compartments may share structure and behavior.
26. Compartments have their own identity.

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

named places in associations without taking attributes or behavior into account. On the other hand, programming languages such as ObjectTeams/Java (OT/J) [20] allow for dynamically adapting the behavior of objects at runtime. To provide a universal formal role modeling language, Steimann surveyed approaches until the year 2000 [47, 48]. Based on this, 15 different questions on the term role were identified. Some questions are conflicting; it seems that no role-based programming language will ever be able to realize all of those features. Another survey analyzed role-based modeling and programming approaches between the years 2000 and 2014 [29]. An overview of the identified questions of Steimann and Kühn et al. is given in Table 2.1. The authors in [29] focus more on the three different aspects roles try to serve, namely their *relational*, *contextual*, and *behavioral* nature. The relational aspect denotes that different entities interact with each other, or are connected by using roles. With the contextual aspect, roles may be utilized to describe context-dependent features of entities. Finally, roles address behavioral aspects of entities, which refer to the dynamic set of attributes as well as to methods.

It is necessary to further investigate the heavily overloaded term *context*. Context can be understood as environmental information and as objectified collaboration containing other entities. A context surrounds an entity and provides additional information. Information may be time, place, temperature, or the state of the application running. Paradoxically, even the lack of information about an entity can be regarded as an information. Furthermore, entities are always attached to a specific context. For instance, sensor data (like GPS) is only valid for a certain device and its user. For other users not currently present, this information may not be relevant. In sum, this general context definition describes an entity's environmental information that has no specific identity, no intrinsic behavior and is omnipresent. In contrast, a *compartment* as introduced by [29] is defined as: *"[W]ithin modeling languages, context represents a collaboration or container of a fixed, limited scope. To overcome this dichotomy, researchers avoided the term context by using other terms, i.e., environments, institutions, teams, and ensembles. In turn, we use the term compartment as a generalization of these terms to denote an objectified collaboration with a limited number of participating roles and a fixed scope."* [29, p. 146]. While a context (e.g., a cold and rainy day in London) is intentional (described by rules or attributes), without its own identity, intrinsic behavior or existential parts and with an indefinite lifetime - a compartment (e.g., a first-class train car) is extensional, i.e., is explicitly specified. Its instances carry identity, have behavior, state, a defined lifetime and contain roles as its parts. Hence, a compartment can be seen as an objectified collaboration for the contained roles. This thesis is based on this definition of compartments, as introduced by CROM.

Contemporary literature has not been able to provide a unique definition of what a role is, especially with regard to runtime. These aspects of the semantics of the role concept have been described in a variability analysis [17]. This analysis relies on the encountered semantics of roles which goes far beyond the analysis presented in [29], because many runtime features are investigated. We derived a list of features for roles at runtime. This list (see Table 2.1) permits us to reflect over the set of features *SCROLL* covers, which is also used during the evaluation.

2.2 FOUNDATIONS OF DISPATCH

Many developed programming languages offer support for advanced modularization mechanisms, like in the context of this thesis with roles, but are implemented as transformations to the imperative intermediate representation of an already established language. Their core constructs largely overlap in semantics [28]. Hence, reusing the corresponding transformations requires reusing their syntax as well, which is too limiting.

With *SCROLL*, we identified dispatching as fundamental to role-based programming and propose a declarative and parameterizable approach for four-dimensional, context-aware dispatch at runtime. To increase the modularity of programs, research has introduced different abstraction mechanisms, where one concrete program module does not refer to another concrete module,

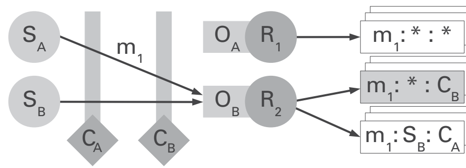

Figure 2.2: The concept of four-dimensional dispatch (with methods m_n, sender S_n, contexts C_n, and objects O_n with roles R_n) [22].

but only abstractly specifies the functionality or data to be used (*polymorphism*). With roles, this principle is pushed even further. The mechanisms for polymorphism are manifold; they include traditional receiver-type polymorphism and reach out to multiple and predicate dispatching [12], pointcut-advice in aspect-oriented programming [34], or layered methods in context-oriented programming [22]. Those languages typically overlap in their semantics but differ syntactically. Compiler frameworks [9, 1] only support reusing the implementation of a language's execution semantics if that language is extended syntactically.

The process of *dispatching* resolves abstractions and binds concrete functionality to their usage [6]. This declarative mechanism determines the code to be executed upon a method invocation. It takes place whenever a specific code location is referenced during the program execution. A well-known example of dispatching is *receiver-type polymorphism*. Here, dispatch is choosing the method implementation based on the dynamic type of the receiver object. Languages that go beyond such classic receiver-type polymorphism are called *advanced-dispatching languages*, as they compose functionality in different, more flexible ways and incorporate additional runtime state. Hence, role-based dispatch can be seen as an advanced dispatch suitable to overcome the subtle ambiguities introduced with the concept of role-playing objects. However, role dispatching is a dynamic process. Thus, techniques solely extending the static program structure cannot satisfactorily realize this dynamic process. Implementations of role-based languages often build on the back-ends of an already established language, thereby reusing the implementation of the constructs in its intermediate language. But not all constructs of role-based languages have a trivial mapping to the established intermediate language (e.g., Java bytecode). The resulting semantic gap between source and intermediate language, i.e., the inability of the intermediate language to directly express the new language's mechanisms, requires compiling the language's high-level concepts down to low-level imperative code. This was considered as inappropriate during the development of *SCROLL*, since building and maintaining a whole new compiler tool chain is too time-consuming and out of the scope of this thesis.

Advanced compiler frameworks could have been assisting in this task, and even enable to reuse the non-trivial code generation for role-specific language constructs that have no direct counterpart in the target intermediate language. But this reuse requires the new language to be a syntactic extension of an existing one. While code transformations defined on the common intermediate language are shared among all language extensions, they cannot exploit knowledge about new source language constructs. This knowledge is lost during the transformation to the intermediate language. With that, existing tools' usefulness is greatly reduced. The developer has to observe the program execution in terms of the generated, imperative code in the intermediate language rather than in terms of the new language's source-level abstractions. This is considered as a major drawback for existing role-based programming languages. Experimenting with their specific dispatch implementations (e.g., changing and adapting its semantics at runtime) is not possible at all. Thus, we decided to use a library approach with *SCROLL* for role-based, dynamic dispatch, making custom compiler and code generation unnecessary. In the following, the concept of **multi-dimensional message dispatch** [22] in the context of role-playing objects is presented. It is intended to help the reader to understand its basics for the remainder of this thesis. This can be derived from:

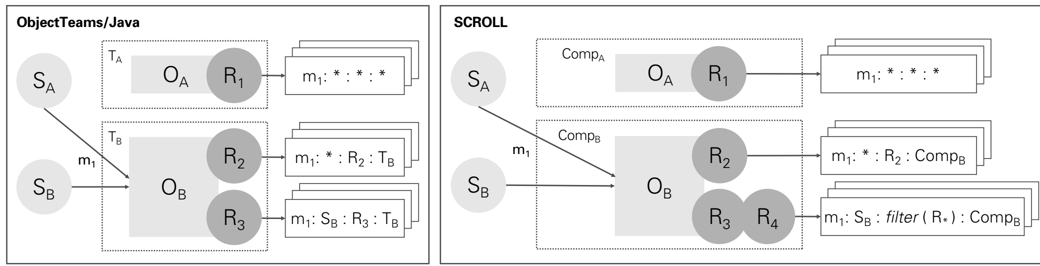

Figure 2.3: Multi-dimensional Dispatch with OT/J and *SCROLL* in comparison (with methods m_n, sender S_n, teams T_n, compartments $Comp_n$, and objects O_n with roles R_n).

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

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

Three-dimensional dispatch For instance, subjective programming [46] extends the object-oriented method dispatch by yet another dimension. With that, the targeted method is not only selected in dependence of its name and receiver, but also in dependence of the sender.

Four-dimensional dispatch As part of this thesis, the fourth layer is put on top of the dispatching concepts, inspired by subjective and context-oriented programming [22]. Now also the context of the actual message send, hence the overall system's context, is taken into account (see Fig. 2.2). Based on that information, methods or their partial definitions are selected or excluded from the message dispatch. This finally enables the context-dependent behavioral adaptation and variation needed for role-based programming.

To provide a more fine-grained dispatch, role-oriented programming languages, such as, OT/J, encapsulate the fourth dimension within contexts implemented as first-class citizens (e.g., in OT/J these contexts are called Teams). With that, the actual method dispatch is configurable at compile-time for a predefined set of role types (see Fig. 2.3 left side). *SCROLL* builds conceptually on-top of that. It allows for the dynamic attachment of arbitrary many roles at runtime, still encapsulated in first-class contexts, called compartments, as already introduced (see Sect. 2.1). Now, the method dispatch can be re-configured at runtime via filtering the set of all attached roles steered by user-defined functions (see Fig. 2.2 right side).

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

2.3 FOUNDATIONS OF GRAPHS AND GRAPH FILTERING

Graphs are one of the most important data structures in programming and computer science in general. They appear in almost all applications. Structures linked with software models, pointers, object nets, databases, and with various schemes, are in essence graphs. A labeled graph is a

Table 2.1: Notation overview for graph traversals and filters.

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

collection of objects (nodes or vertices) which are connected via linking objects (edges). Nodes and edges may be associated with names, additionally (called labels). Graphs are applied as background data structure in *SCROLL*. The real power of graphs makes itself apparent when traversing multiple steps in order to unite disparate not directly connected vertices by a path. The type of path taken, defines the higher order, inferred relationship that exists between two vertices. Paths form the core of the presented graph traversals. A traversal refers to visiting elements (i.e., vertices and edges) in a graph in some algorithmic fashion. Exactly those efficient graph operations yield an unconventional problem-solving style. This style of interaction is dubbed the graph traversals in the following and forms the primary point of discussion for this section.

The functional, flow-based approach to traversing graphs and different types of traversals over different types of graph data sets, supports different types of problem solving processes [45]. The most primitive, read-based operation on a graph is a single step traversal from element i to element j, where $i, j \in (V \cup E)$. For example, a single step operation can answer questions such as “which edges are outgoing from this vertex?” or “which vertex is at the source of this edge?”. Single step operations expose explicit adjacencies in the graph. Various types of those single step traversals can be found in Table 2.1. These operations are defined over power multiset domains and ranges. This naturally allows for function composition. When edges are labeled and elements have properties, it is desirable to constrain the traversal to edges of a particular label or elements with particular properties. These operations are known as *filters* and are abstractly defined in Table 2.1 as well. Through function composition of single step traversals, we can define graph traversals parameterized by filter functions which is suitable to answer questions, such as, “which roles is the current object playing?”.

2.4 PROBLEMS AND RESEARCH CHALLENGES

For the implementation of adaptive systems, the software design needs to contain definitions and descriptions to handle the static and dynamic aspects of problems in its domain, namely, for business logic as well as adaptation logic. The interactions of the participating objects with the client are non-trivial. Each of those clients will handle the instances differently and with different use-cases in mind, e.g., handling different kinds of robots and their individual work plans. This interaction needs to be described and constrained from the viewpoint of all relevant contexts. Furthermore, a sound understanding of object collaborations and relationships at runtime is crucial. When developing an adaptive software system, the programmer must be able to define those collaborations. Once they are understood, it is important to group and separate them for the sake of comprehensibility and reuse. Relationships need to be assigned to relevant contexts. Derived from the two aforementioned tasks, this reuse is one of the most important attributes adaptivity on the resulting system benefits from. With invariants and constraints, the behavior of the objects working together in that systems can be restricted and checked during runtime. Finally, it is impossible to foresee each and every possible future use-case and context. Handling the adaptation based on various events with regard to the current system context and state leads to *if-bloating* with nesting and inter-tangling of business and adaptation logic. The problems mentioned above manifest themselves as:

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

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

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

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

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

Applying the concept of roles now enables explicit separation of concerns of the relationships at instance level. This cannot be achieved with normal class interfaces. A role type exactly specifies how the instance of a class interacts in a certain context. The major part of the complexity of those adaptive software infrastructures stems from complex object collaborations. These collaborations become manageable with the break-down into individual, smaller role models. Each role model describes an individual aspect of the object collaboration and adaptation. With the resulting clear separation of object collaboration into smaller pieces with regard to the concerns of the problem space and their composibility, a high amount of reuse is enabled. Furthermore, a role type is a good target for invariants and constraints. And finally, adding role types dynamically allows for the adaptation of unforeseen contexts. The expressiveness of the concept of roles require a whole new level of dispatching semantics, i.e., a dispatch that needs to support context-awareness and dynamically evolving objects on the instance level.

The following research challenges and requirements can be derived from the aforementioned problems. We aim for a solution that requires no additional tooling (/F.1/). As almost all the contemporary approaches for role-based programming use custom compilers or code generators, they break with existing tool-chains (e.g., debuggers) and cannot be used with ease in widely established

Table 2.2: Functional and non-functional requirements for *SCROLL*.

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

integrated development environments (e.g., Eclipse or IntelliJ). This hinders maintainability and extendability and often results in abandoned projects. With the notion of roles, their expressiveness and their subtle ambiguities, the resulting dispatch semantics needs to be handled explicitly and at runtime (/F.2/). Hence, the solution developed during this thesis should support this new kind of context-aware dispatch on the instance level for dynamically evolving role-playing objects. As this context-aware dispatch introduces additional dimensions, those must be addressed as well (/F.3/). Thus, on-top of associating the method with a name and tailoring the dispatch to the receiver or sender context, the overall system context needs to be addressed additionally. And finally as a fourth requirement, a maximum of the features of roles in role-based programming has to be supported by the developed implementation to increase modularity (/F.4/). To make the aforementioned dispatch implementation as usable and attractive to the developer it should be declarative and parameterizable at runtime (/S.1/). This offers high flexibility for context-aware adaptation and additional separation of concerns. To support the application programmer even better, the programming model and API should be easy to use and readable even for inexperienced developers (/S.2/). A reasonable performance and scalability of the implementation is important as well for real-world scenarios and show its practical applicability (/S.3/). As a follow-up to /F.1/, code should be easily manageable by existing tool-chains so that future role researchers, i.e., researchers interested in role-based programming, and application developers can continue the development and provide extensions and adaptation for future use-cases and scenarios (/S.4/). For easier integration of existing software systems, we do not want to impose additional burden to the developer writing adapters, proxies, or management code to integrate existing legacy code (/S.5/). Finally, and as a result of /F.1/, /S.2/, /S.4/, and /S.5/, the reference implementation developed during this thesis for role-based programming and dispatch should be highly maintainable and extendable so that future researchers have an easy time providing modifications for new use-cases and scenarios (/S.6/, and /S.7/). A summary can be found in Table 2.2.

3 THE EMBEDDED DSL *SCROLL*

Scripting languages like Python, JavaScript, Ruby, Perl or Lua offer a flexible object semantic to the developer. On the one hand, programmers can rely on classical object-oriented features, such as inheritance, encapsulation and polymorphism, and on the other, they are able to add and remove members from existing objects or merge them at any given point in their life-cycle [35] which is usually not available in statically typed object-oriented languages. Unfortunately, using inheritance, mixins and traits or adapting design-patterns has many disadvantages. The first three techniques will result in a very static system design and exponentially many classes, while the use of patterns often leads to split-objects and the need of additional management code. Adding and removing members from existing objects at runtime are indeed very useful operations for modern software-systems that have a very high demand for adaptivity and need to cope with complexity and change [14]. Is bridging the gap between statically-typed, object-oriented languages and roles as evolving objects at runtime possible without too much effort? The main contributions of this work permit us to answer this questions positively:

SCROLL and the SCROLL MOP An overview on *SCROLL*, an embedded DSL and its underlying MOP [26, 36] that allows for the pure embedding [23] of roles in the modern, statically typed object-oriented language Scala. It solely utilizes features that are available through the standard compiler. The library allows for easy integration of legacy code and a high separation of concerns. It is limited with regard to type-safety as one might expect. Nevertheless, having a statically-typed host language for roles supports the developer with the best of both worlds: static typing leads to an earlier detection of programming mistakes through static code analysis, better documentation in form of type-signatures, compiler-optimization, runtime-efficiency and an improved design-time development experience, while the latter supports easy prototyping, change to unknown requirements or unpredictable data and application integration. Essentially, two user groups for *SCROLL* can be identified: the *end-user* (the programmer writing domain-specific, role-based applications), and the *library developer* (adapting or transferring the *SCROLL* MOP and its semantics to his research area).

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

Examples Finally, an example application shows how roles are realized with *SCROLL*.

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

3.1 THE BASIC INGREDIENTS OF *SCROLL*

SCROLL is an embedded method-call interception DSL [33, 36] tailored to the features needed to implement roles and resolve the ambiguities arising with regard to dynamic dispatch. The library approach together with an implementation with Scala was chosen for mainly the following reasons: it allows focusing on role semantics, supports a customizable, dynamic dispatch at runtime, and allows for a terse, flexible representation. No additional tooling (like a custom lexer, parser or compiler) is needed to execute the *SCROLL* MOP. It is purely embedded in the host language, thus uses the standard Scala compiler to generate Java Virtual Machine (JVM) bytecode. With that, the implementation is reasonable small (~1400 lines of code) and maintainable. The

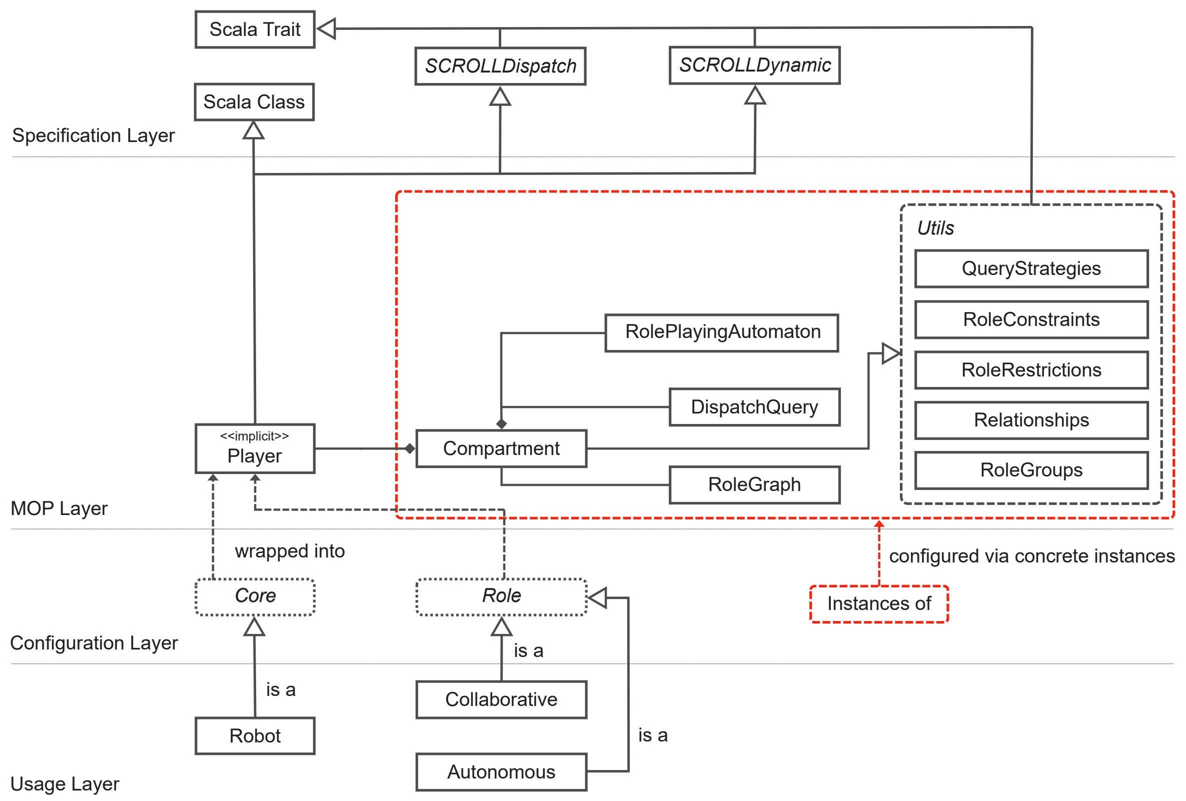

Figure 3.1: An overview of the *SCROLL* metamodel and MOP layers.

programming interface with Scala's flexible syntax holds the property of being easily readable, even to inexperienced users. We have taken a layered approach (see Fig. 3.1) for designing and implementing *SCROLL*:

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

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

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

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

To provide a DSL for the pure embedding of roles in structured contexts, *SCROLL* requires the following basic implementation concepts from the host language (see Fig. 3.1):

Compiler rewrites A concept for compiler rewrites for method calls, functions calls, and attribute access is required. It hands over calls to the library for finding behavior and structure that is

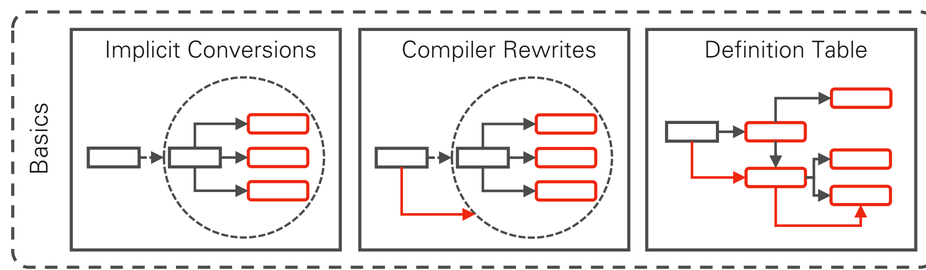

Figure 3.2: Required basics for the implementation of a DSL for roles in structured contexts at runtime.

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

Listing 3.1: Compiler rewrite rules from the `Dynamic` trait [10].

not natively available at the core object. This can be seen as a compiler-supported variant of method-call interception [33].

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

Definition table for the plays relationship The relationships between each individual core object and its roles need to be stored. A *definition table* holds all kinds of program components, whose attributes are created by declaration: types, variables, methods, functions, and parameters [51]. In *SCROLL*, a definition table for roles is implemented with a graph-based data structure, but it may be implemented with tables, maps, or lists as well.

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

3.1.1 THE DYNAMIC TRAIT WITH COMPILER REWRITE RULES

Behavior and state of roles that is not natively available in the core object needs to be addressed somehow. Scala's `Dynamic` trait can be used to implement that behavior [11]. To get invoked, the proper role has to be identified and selected. To do so, calls to role-specific functionality that would normally fail during type checking phase, are rewritten by the compiler according to the rules shown in Listing 3.1. This transformation is type-unsafe, because the actual set of roles as dynamic extensions that are bound to the core object, is not statically known. Hence, static type-safety is not available. *SCROLL* hooks into those rewritten calls and triggers the actual invocation of the appropriate roles, as well as the error handling. It refrains from using runtime exceptions or similar exception-based error handling in case of not being able to find the functionality the developer

is querying for. Instead, Scala's `Either` container type is applied. It has two sub-types, `Left` and `Right`. If an `Either[A,B]` object contains an instance of `A`, then the `Either` is a `Left`. Otherwise, it contains an instance of `B` and it is a `Right`. By convention, it is used to carry the error case as `Left` (e.g., `DynamicBehaviorNotFound`), whereas the `Right` contains the success value (e.g., the result of executing the dynamic behavior). Together with a sealed type hierarchy with data types using case classes that represent errors, very readable messages compared to actual stack-traces from standard Java exceptions are generated.

3.1.2 BOXING WITH IMPLICITS

We want to be able to add roles to any given object of any type in Scala. Implicit conversions [38] provide a lightweight way to expose *SCROLL* 's API for adding, removing and transferring behavior or state to any object and is implemented via the class `Player` from the *SCROLL* MOP layer. Scala's implicit conversion is used to wrap the core object into an equivalent compound object exposing the required API in a type-safe manner. Furthermore, the issue of object schizophrenia needs to be addressed with a clear notion of object identity. The identity of an object should be the same independent of which role is attached. In summary, four kinds of equality tests between pairs of objects (i.e., the core object C and its role instances R_n) are possible:

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

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

1. $C + R_1 == C$: When the equality for a core object playing a role compared to itself is requested, then the compound object (a `Player` instance) maps `equals` to the implementation of the core object.
2. $C + R_1 == C + R_1$: Same as case one, but the right-hand operator of == is a role. Here, the comparison will be done with this role's core object.
3. $C + R_1 == C + R_2$: Same as case three.
4. $C == C + R_1$: We cannot modify the `equals`-method of arbitrary objects using a library approach. If the comparison of a plain core object is required, the `+-Operator` needs to be applied. This will trigger the dynamic conversion using the implicit class `Player` and applies the desired comparison, as in cases one to three.

3.1.3 THE DEFINITION TABLE FOR THE PLAYS RELATIONSHIP

In *SCROLL*, a graph-based data structure is used for implementing the definition table storing the relationships between core objects and its role instances. The role-play graph allows for easy querying of role-specific behavior that was attached to the core object at some point in time. As an implementation, any appropriate graph library can be used. For *SCROLL*, Guava's graph data structure [15] was chosen as underlying graph library already providing the necessary graph-theoretic objects like edge- and node-types as well as simple algorithms for traversing the graph. *SCROLL* makes it easy to plug-in any other convenient library, e.g., for easy scaling or distribution. Additionally, access to roles is cached speeding up the querying for the appropriate structure and behavior hidden in a role. Graph traversals are used and mapped directly to Scala functions.

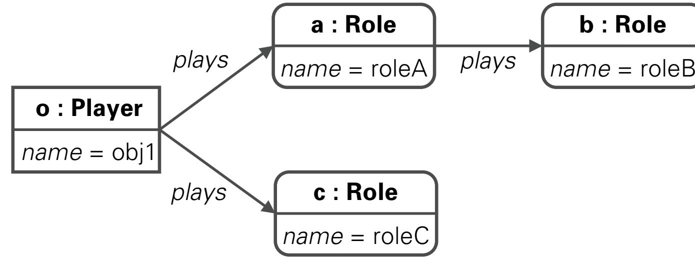

Figure 3.3: Example of a simple role-play graph.

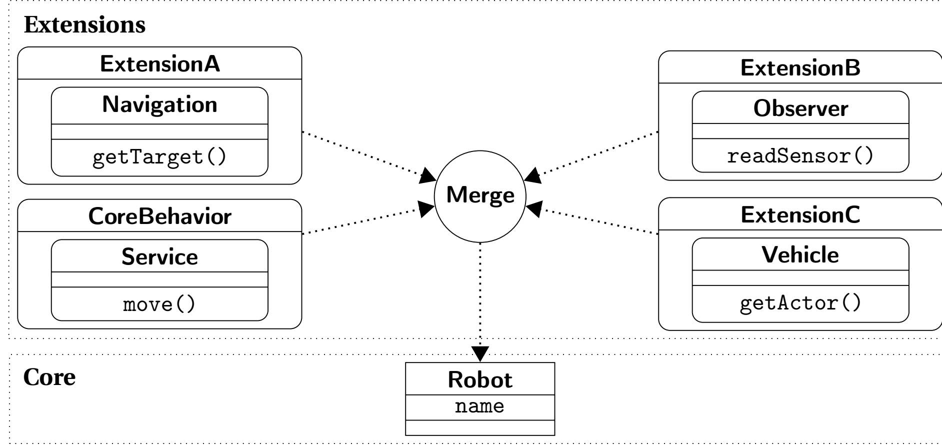

Figure 3.4: Class `Robot` is constructed (dotted arrows) from different roles and acquires the contained behavior.

Consider the example provided in Fig. 3.3. A player type is instantiated (`o`) and plays the role type instances with the property name `roleA`, `roleB` (as deep role), and `roleC`. If i is the vertex representing the object and

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

where

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

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

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

3.2 *SCROLL* BY EXAMPLE

This subsection explains the basic usage of *SCROLL* for the pure embedding of roles. We start with a brief introduction how one can use roles by example (see Fig. 3.4). A standard Scala case class (`Robot`) should be augmented with new behavior encapsulated in three different classes as extensions (`ExtensionA`, `ExtensionB` and `ExtensionC`). Each of them provides a new aspect of the robot via functions, such as, finding a target to move to, or observing sensor values, attached to case classes. This allows for a high degree of separation of concerns with multiple hierarchically structured compartments. The core behavior (with case class `Service`) aggregates all the provided functionality without having to worry about which role delivers which service.

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

Listing 3.2: A naive solution for the robot example. It fails during compilation because `name()` (Line 4) is not available at instances of `Service`.

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

Listing 3.3: A new solution for the robot example using the basic *SCROLL* API.

We now step-wise construct the example. First, only the `name` attribute of the robot should be printed. This naive solution, non-surprisingly, fails during compilation because `name()` (Line 4 in Listing 3.2) is not available at instances of `Service`. To solve this problem of adding behavior dynamically, we now apply the most basic concepts of the *SCROLL* DSL, namely a `Compartment` (Line 2 in Listing 3.3), the +-operator (Line 5 in Listing 3.3), and the `play` API call (Line 9 in Listing 3.3). The `Compartment` trait exposes *SCROLL* 's basic API to the current class, allowing the programmer to use the +-operator, and the `play` method. Because any given object should be allowed to play roles, we cannot assume that this object actually provides the +-operator. Thus, Scala's implicit conversion [38] is used to wrap the core object into an equivalent compound object exposing the required API as mentioned above. By calling the +-operator, applying implicit lifting, the user is able to forward arbitrary calls to some roles he assumes should be available on the core object without worrying about their actual location. Calling `play` adds a play relationship between a player (instances of `Robot`) and a role instance (instances of `Service`), finally enabling the call to `name()` (Line 5 in Listing 3.3).

As a final step, for better separation of concerns, new functionality from roles is now grouped into extensions represented by individual compartments, e.g., with the compartment `ExtensionA` (Line 11 in Listing 3.4). The role-playing graph, holding the relationships between role-playing objects (e.g., instances of `Robot`) and their roles (e.g., instances of `Service`, and `Navigation`), is defined compartment-wise. Hence, in the anonymously instantiated compartment at Line 16 in Listing 3.4, making the robot actually move, those individual role-playing graphs are merged into a new one, spanning now multiple compartment instances. With that, the role-playing relationships defined in the anonymously instantiated compartment are now a part of those within `CoreBehavior`, and `ExtensionA`, respectively. Hence, all the requested behavior (i.e., `name()`, `getTarget()`, and `move()`) is available. The full example can be found in Listing 3.5.

3.3 TECHNICAL LIMITATIONS

SCROLL allows for role-based programming with the concept of dynamically evolving objects and purely embeds roles in a statically typed, object-oriented host language. This supports the developer with the best of both worlds: static typing leads to an earlier detection of programming mistakes through static code analysis, better documentation in form of type-signatures, compiler-

```
case class Robot(name: String)
object CoreBehavior extends Compartment {
  case class Service() {
    def move() {
      val name: String = +this name()
      val target: String = +this getTarget()
      info(s"$name moves to $target.")
    }
  }
}
object ExtensionA extends Compartment {
  case class Navigation() {
    def getTarget = "kitchen"
  }
}
new Compartment {
  val robot = Robot("Pete") play Service() play Navigation()
  ExtensionA partOf CoreBehavior partOf this
  robot move()
}
```

Listing 3.4: The third solution for the robot example using the more advanced *SCROLL* API.

```
case class Robot(name: String)
object CoreBehavior extends Compartment {
  case class Service() {
    def move() {
      val name: String = +this name()
      val target: String = +this getTarget()
      val sensorValue: Int = +this readSensor()
      val actor: String = +this getActor()
      info(s"$name moves to $target with $actor and sensor value of $sensorValue.")
    }
  }
}
object ExtensionA extends Compartment {
  case class Navigation() {
    def getTarget = "kitchen"
  }
}
object ExtensionB extends Compartment {
  case class Observer() {
    def readSensor = 100
  }
}
object ExtensionC extends Compartment {
  case class Vehicle() {
    def getActor = "wheels"
  }
}
new Compartment {
  val myRobot = Robot("Pete") play Service() play Navigation() play Observer() play
    Vehicle()
  ExtensionC partOf ExtensionB partOf ExtensionA partOf CoreBehavior partOf this
  myRobot move()
}
```

Listing 3.5: The full `RobotExample` source code.

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

Listing 3.6: The `RobotExample` console output.

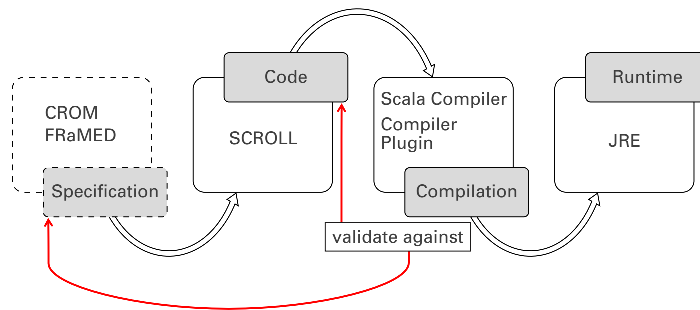

Figure 3.5: The CompilerPlugin toolchain.

optimization, runtime-efficiency and an improved design-time development experience, while dynamic objects support easy prototyping, change to unknown requirements or unpredictable data and application integration. Nevertheless, implemented as a library approach on-top of the Scala programming language, there exists no built-in abstraction of those dynamically evolving objects on type level yet. Hence, the following limitations apply.

SCROLL uses Scala's `Dynamic` trait [11] to address all dynamic behavior from roles that is not available at the core object. This is comparable to the usual implementations of dispatch tables (e.g., with C++ vtable, or Java call-sites). Calls to role-specific functionality that would normally fail during type checking phase of Scala are rewritten *after* the typing phase of the Scala compiler. At this point, type-safety is lost. The actual set of roles as dynamic extensions that are bound to the core object is not statically known, hence static type-safety is not available. At runtime, compound objects representing role-playing, dynamic objects are always represented as `Player[T]`, where `T` refers to the type of the core object. No special typing construct is available to mirror role-playing objects in first place, hence calls via the `Dynamic` trait cannot be statically typed. To remedy this shortcoming, and to help the developer, providing additional warnings and error messages whenever the requested dynamic behavior is unlikely to exist at all, the *SCROLLCompilerPlugin* was developed (see Sect. 3.4).

Furthermore, as the quantitative evaluation shows, *SCROLL* performs quite slowly due to the heavy use of the Java Reflection API. In the scope of this theses that is not to be considered critical, as it serves as a testbed for dynamic dispatch. Implementing suitable, role-aware types (`invokedynamic` on the JVM), use Scala macros, or implement a Scala compiler plugin would improve the overall performance. This is considered to be out of the scope for this thesis, but targeted as future work.

3.4 THE *SCROLL* COMPILER PLUGIN

As soon as the compiler triggers its rewrite rules (Scala's `Dynamic` as explained in Sect. 3.1.1) certain type-safety is lost because it cannot be statically determined if a role is actually bound during runtime. FRaMED [27] is able to export instances of CROM [29] as Ecore files serialized as XMI. With the help of such an optionally imported file the *SCROLLCompilerPlugin* [30] will check all the statements invoking role calls against the available player and role classes from the model instance and the class definitions in the current scope. The SCROLLCompilerPlugin generates meaningful messages and reports them as warnings or compile time errors (which is configurable) to the developer. To gather the behavior offered in all possibly attached roles as dynamic extensions, all relevant binding- and unbinding statements, player classes and their behavior, and all calls to the `Dynamic` trait, the Scala Abstract Syntax Tree (AST) is traversed at compile-time right after the typer phase of the standard Scala compiler [32]. The algorithm used is a program analysis which runs on the AST until all AST subtrees are covered. After running this program analysis at compile-time, the sets of statements and collected behavior are compared against each other and warnings or errors will be reported to the user. An overview of the resulting tool chain is visualized in Fig. 3.5.

4 EVALUATION

The evaluation for *SCROLL* is split into four parts. Firstly, we analyzed the fulfillment of the requirements stated in Sect. 2.4. Secondly, *SCROLL* was analyzed based on a previously defined classification scheme [29]. Then, the variability analysis from [17] was applied. Finally, we benchmarked various implementations for roles at runtime and identified performance bottlenecks of *SCROLL*. In sum, these are the results:

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

Feature-based analysis of SCROLL To investigate how well the implementation with *SCROLL* blends into contemporary approaches, the previously defined scheme from [29] with 26 classifying features of roles was applied. *SCROLL* fully implements 22 of them (see Table 4.1).

Summary for runtime feature analysis This investigates the role semantics by a feature analysis loosely based on [17]. Instances of *classes* as the fundamental basis of roles in *SCROLL* with their corresponding role-playing constraints and supertype restrictions are fully incorporated. Furthermore, many *constraints* with regard to the cardinalities imposed on the player as well as the role side are supported. *Relationships*, e.g., with the concept of inheritance, can be handled and most of the well-known *properties* (e.g., static methods, class methods and fields) are available within *SCROLL*. In addition, the analysis for role-specific *behavior* reveals *SCROLL* 's ability to dispatch calls on various entities (e.g., roles and its players, the notion of self, and super). The notion of *identity* is discussed with the question in mind if roles have an unique identity or it is rather shared between a role and its player. When it comes to handling the *life cycle* of roles, *SCROLL* offers support for a fairly simple implementation of role creation, attachment, movement and removal. Finally, *type* related issues are discussed. As a result, it was shown that *SCROLL* realizes a good balance for the role and compartment concepts with regard to statically and dynamically languages.

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

5 CONCLUSION AND FUTURE WORK

In the modern world, software systems are expected to adapt to a changing environment as they become more and more ubiquitous. During their lifetime, new features are requested, existing requirements change, and the hardware and operating systems are regularly being renewed. Software written for a specific purpose may become useful in situations and environments, which the developer did not dare to anticipate. Those situations are ubiquitous in the physical world (e.g., on wearables and smartphones) and ubiquitous in the software world of Internet-based applications. Object-oriented programming, as being widely used to build extensible and flexible software systems, is successful, because it supports programming with data structures that closely resemble the

Table 4.1: Comparison of coeval approaches for etablishing roles at runtime based on 26 classifying features extracted from [29] presented in Table 2.1. It differentiates between fully (■), partly (⊞), and unsupported (□) features.

Feature [29]	**Chameleon** [39]	**OT/J** [21]	**Rava** [19]	**powerJava** [50]	**Rumer** [3]	**ScalaRoles** [42]	**NextEJ** [24]	**JavaStage** [4]	**SCROLL** [31]
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.	□	■	□	⊞	■	■	■	□	■

problem domain. However, future software systems expect a higher level of dynamism, not offered by pure object-oriented concepts. With dynamically typed, object-oriented scripting languages, a very flexible programming style is available. Modules, classes and objects can be extended arbitrarily at runtime. But programming in a dynamically typed language comes at a cost. Without static type information, it is impossible to analyze programs statically and catch whole classes of programming errors before actually running the program. The burden is solely carried by the programmer. To cope with challenges imposed by ubiquitous, highly adaptive software systems, researchers proposed several approaches, including the language concept of roles. This concept allows for extracting the context-dependent behavior from the classes and model it in separate role types. Together with role-based, dynamic dispatch, a new level of separation of concerns is revealed. The core behavior and structure is defined in the object's type. Context-dependent and evolving parts are then specified in role types. Role-playing objects are able to start and stop playing roles to adapt their behavior and structure dynamically during runtime, without the need for reinstantiation. However, role-playing objects need specific forms of multi-dispatch.

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

***SCROLL* and the *SCROLL* MOP** This thesis presents *SCROLL*, an embedded method-call interception DSL as library with its underlying MOP [33, 36, 26] that allows for pure embedding [23] of roles in a modern, statically typed object-oriented language (Scala) without changing its syntax. It solely utilizes features that are available through the standard compiler. This library allows for easy integration of legacy code and a high separation of concerns. It has a minimalistic design and stays below 1400 lines of code.

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

A simple implementation pattern for roles in structured contexts The implementation pattern behind *SCROLL* requires three basic components, namely, compiler rewrites, i.e., a compiler-supported variant of method-call interception [33], implicit conversions for assembling a compound object from the core plus all of its roles, and a definition table of the relationships between each core object and its roles, the role-play graph.

A role-based dispatch configurable at runtime A declarative and parameterizable approach for four-dimensional, role-based dispatch at runtime is presented. This enables the developer to overcome the subtle ambiguities with roles in structured contexts by utilizing an explicit representation of dispatch rules as function objects [49]. The dispatch is based on four dimensions: the name of the computational unit, the context of the receiver, the context of the sender, and, for the first time, on structured contexts. The dispatch can be configured dynamically by node filter functions.

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

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

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

As every novel approach in the field of programming language design and implementation, *SCROLL* opens a wide space for future work. Several developments are currently work in progress or targeted for investigation in the near future. In **interdisciplinary collaborations**, we aim for other use-cases for the concept of dynamically evolving objects. They should help the domain expert to cope with domain-specific implementation concerns. Specifically in systems biology and, more generally, in scientific computing (e.g., with a Next-Generation Parallel Particle-Mesh Language [25]), using *SCROLL* looks promising. With respect to the required performance, **optimizations** for translating the specific binding and behavior-lookup for dynamic objects need to be developed. A promising direction is the investigation of the `invokedynamic` bytecode keyword introduced with Java 7 to provide an alternative implementation of *SCROLL*. Furthermore, **other**

dynamic objects, like facets, parts, phases, and aspects could be investigated whether they can be integrated into *SCROLL*. With more case studies, it needs to be investigated if the proposed dynamic, role-based dispatch is **expressive** enough to cope with the requirements of context adaptation. Is a mapping to, e.g., predicate dispatch feasible? What are the benefits, when translating this dispatch semantics into a new role- and context-aware type system? Are existing type systems (e.g., dependent type systems) sufficient? Finally, in [37], the authors provide **metrics for dispatch** (e.g., dispatch ratio, choice ratio, or degree of dispatch). These metrics focus on method definitions and can be measured statically. Tailored to the notion of roles, one could investigate the degree of adaptability provided by the *SCROLL* dispatch concept in comparison to existing approaches.

BIBLIOGRAPHY

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I*, pages 293–334. Springer, 2006.

[2] Charles W. Bachman and Manilal Daya. The Role Concept in Data Models. In *Proceedings of the Third International Conference on Very Large Data Bases,*, pages 464–476, Tokyo, Japan, 1977.

[3] Stephanie Balzer, Thomas Gross, and Patrick Eugster. A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 323–346. Springer, 2007.

[4] FSRBM Barbosa and Ademar Aguiar. Modeling and programming with roles: introducing JavaStage. Technical report, Instituto Politécnico de Castelo Branco, 2012.

[5] Daniel G Bobrow, Richard P Gabriel, and Jon L White. Clos in Context - The Shape of the Design Space. *Object Oriented Programming: The CLOS Perspective*, pages 29–61, 1993.

[6] Christoph Bockisch, Andreas Sewe, Haihan Yin, Mira Mezini, and Mehmet Aksit. An In-Depth Look at ALIA4J. *Journal of Object Technology*, pages 7:1–28, 2012.

[7] Guido Boella, Steffen Goebel, Friedrich Steimann, Steffen Zschaler, and Michael Cebulla. 2'nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies, 2007.

[8] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '92, pages 201–217, New York, NY, USA, 1992. ACM.

[9] Torbjörn Ekman and Görel Hedin. The jastAdd Extensible Java Compiler. *ACM Sigplan Notices*, 42:1–18, 2007.

[10] EPFL. Scala Dynamic Trait ScalaDoc, 2016. Accessed: 1st December 2016, 09.00.

[11] EPFL. Scala Dynamic Trait SIP, 2016. Accessed: 1st December 2016, 09.00.

[12] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conference on Object-Oriented Programming*, pages 186–211. Springer, 1998.

[13] Martin Fowler. Dealing with roles. In *Proceedings of PLoP*, volume 97, 1997.

[14] J. Frank Furrer. Zukunftsfähige Softwaresysteme. *Informatik-Spektrum*, pages 1–9, 2015.

[15] Google. Guava, 2016. Accessed: 1st December 2016, 09.00.

[16] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems (TOIS)*, 14:268–296, 1996.

[17] Kasper Bilsted Graversen. *The nature of roles.* PhD thesis, PhD thesis:/Kasper Bilsted Graversen.–Copenhagen, IT University of Copenhagen Copenhagen, 2006.

[18] Bill Harrison. Subject-oriented Programming vs. Design Patterns, 2016. Accessed: 1st December 2016, 09.00.

[19] Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. Rava: Designing a Java extension with dynamic object roles. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 7–pp. IEEE, 2006.

[20] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Net.Object Days 2002*, October 2002.

[21] Stephan Herrmann. ObjectTeams/Java. Technical report, AAAI Fall Symposium, 2005.

[22] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7, 2008.

[23] Paul Hudak. Modular Domain Specific Languages and Tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE, 1998.

[24] Tetsuo Kamina and Tetsuo Tamai. Towards safe and flexible object adaptation. In *International Workshop on Context-Oriented Programming*, page 4. ACM, 2009.

[25] Sven Karol, Pietro Incardona, Yaser Afshar, Ivo F Sbalzarini, and Jeronimo Castrillon. Towards a Next-Generation Parallel Particle-Mesh. In van der Storm, Tijs and Erdweg, Sebastian., editor, *Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015)*, volume abs/1508.03536, pages 7–8. van der Storm, Tijs and Erdweg, Sebastian., 2015.

[26] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* mitpress, 1991.

[27] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. FRaMED: Full-fledge Role Modeling Editor (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 132–136, New York, NY, USA, 2016. ACM.

[28] Thomas Kühn and Walter Cazzola. Apples and oranges: Comparing top-down and bottom-up language product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 50–59. ACM, 2016.

[29] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 141–160. Springer International Publishing, 2014.

[30] Max Leuthäuser. SCROLLCompilerPlugin, 2016. Accessed: 08th May 2017, 09.00.

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

[32] Lightbend Inc. Akka FSM, 2016. Accessed: 1st December 2016, 09.00.

[33] Ralf Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55. ACM, 2002.

[34] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *European Conference on Object-Oriented Programming*, pages 2–28. Springer, 2003.

[35] Pottayil Harisanker Menon, Zachary Palmer, Alexander Rozenshteyn, and Scott Smith. Types for flexible objects. Technical report, Technical report, The Johns Hopkins University, 2013.

[36] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37:316–344, 2005.

[37] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *ACM SIGPLAN Notices*, volume 43, pages 563–582. ACM, 2008.

[38] Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala: a comprehensive stepby-step guide. *Artima Inc, August*, 2008.

[39] Kasper Østerbye. Implementation of a role language for object-specific dynamic separation of concerns. In *AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[40] Barbara Pernici. Objects with roles. *ACM SIGOIS Bulletin*, 11(2-3):205–215, 1990.

[41] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM, 2002.

[42] Michael Pradel and Martin Odersky. A Lightweight Approach towards Reusable Collaborations. In *International Conference on Software and Data Technologies (ICSOFT'08)*, 2008.

[43] Python Software Foundation. Python 3 Glossary on Duck-Typing, 2016. Accessed: 1st December 2016, 09.00.

[44] Dirk Riehle. *Framework design*. PhD thesis, Diss. Technische Wissenschaften ETH Zürich, Nr. 13509, 2000, 2000.

[45] Marko A Rodriguez and Peter Neubauer. The graph traversal pattern. *arXiv preprint arXiv:1004.1001*, 2010.

[46] Randall B Smith and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2:161–178, 1996.

[47] Friedrich Steimann. *Formale Modellierung mit Rollen*. PhD thesis, TU Hannover, 2000. Habilitation thesis.

[48] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000.

[49] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education India, 1995.

[50] van der Torre. Roles as a Coordination Construct: Introducing powerJava. *Electr. Notes Theor. Comput. Sci*, 150(1):9–29, 2006.

[51] William M Waite and Gerhard Goos. *Compiler construction*. Springer Science & Business Media, 2012.