# Towards Virtual Machine Support for Contextual Role-Oriented Programming Languages

Lars Schütze
lars.schuetze@tu-dresden.de
TUD Dresden University of Technology
Dresden, Germany

Jeronimo Castrillon
jeronimo.castrillon@tu-dresden.de
TUD Dresden University of Technology
Dresden, Germany

## ABSTRACT

Adaptive software becomes more and more important as computing is increasingly context-dependent. Runtime adaptability can be achieved by dynamically selecting and applying context-specific code. Role-oriented programming has been proposed as a paradigm to enable runtime adaptive software by design. Roles change the objects' behavior at runtime, thus adapting the software to a given context. Most approaches focus on optimizing language implementations neglecting the fact that the generated code is a verbose description of contextual roles in an object-oriented paradigm, which incurs an overhead. This paper takes a novel approach to reduce the semantic gap. We propose ObjectTeams/Truffle, to the best of our knowledge, the first virtual machine that optimizes the dispatch of contextual roles. We evaluate the implementation with a benchmark for role-oriented programming languages achieving a speedup of up to 2.49× over the reference implementation ObjectTeams/Java and 1.2× over an optimized version ObjectTeams/Java using Dispatch Plans.

## CCS CONCEPTS

• **Software and its engineering** → **Software performance**; *Interpreters*; *Context specific languages.*

## KEYWORDS

role-oriented programming, virtual machine, dispatch, quickening

**ACM Reference Format:**
Lars Schütze and Jeronimo Castrillon. 2023. 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), July 17, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605154.3605851

## 1 INTRODUCTION

Separation of concerns is the main technique to conquer the increasing complexity of software, as it allows for decomposing a system into different smaller components. Decomposition is typically dominant, e.g., by object or function, and is predefined by the underlying programming language. Prominent approaches are aspect-oriented programming (AOP) [24], context-oriented programming (COP) [20], and role-oriented programming (ROP) [4, 33, 40]. Role-oriented programming has been proposed as an extension to object-oriented programming to enable adaptive software by design. Classes represent the structural aspect of the domain while roles capture the behavioral aspects. To model context-dependency, compartments encapsulate roles and represent the context in which these roles can be active. Behavioral changes are implemented by objects playing and renouncing roles, which in fact adds and removes behavior to and from the object. ObjectTeams [18] is a mature implementation of contextual roles that provides an overall good performance while supporting most of the features attributed to roles [26]. To further improve execution performance, there exist other language runtimes for ObjectTeams that capture and optimize the dispatch to role functions at runtime [38].

Mapping the role-oriented paradigm into the object-oriented paradigm creates a semantic gap (see Sec. 2.2 for a detailed explanation). Composing methods (i.e., adaptations) out of decompositions and their execution violates assumptions common language implementations hold about lookup resulting in inferior performance [30, 35]. Most approaches neglect that the generated code of the adaptive mechanisms is represented as a verbose description as some mechanisms cannot be directly represented in an object-oriented paradigm, which incurs an overhead [16].

This paper takes a novel approach to reduce the semantic gap. We propose ObjectTeams/Truffle, to the best of our knowledge, the first virtual machine that optimizes the dispatch of contextual roles. The dynamic nature of contextual roles are a perfect candidate to get supported by a virtual machine. ObjectTeams/Truffle is an implementation of the ObjectTeams language model [19] in Espresso, a meta-circular Java bytecode interpreter for the GraalVM [46].

With ObjectTeams/Truffle we could generate a speedup of up to 2.49× (mean 2.23×) compared to ObjectTeams/Java. Compared to ObjectTeams/InvokeDynamic the Role VM is up to 1.22× faster (mean 1.18×).

## 2 BACKGROUND

This section introduces the role-oriented programming concept and ObjectTeams/Java as the most optimized representative. We further explain key concepts used in contemporary high-performance dynamic programming language runtimes.

### 2.1 Role-Oriented Programming

Classes in the object-oriented paradigm are good at capturing structures of a domain but not at capturing varying behavior of objects or groups of objects. The idea of roles originated from the domain of databases, where it was observed that persisted objects tend to

```
1  class Account {
2   void withdraw(float amount) { ··· }
3  }
4  team class Bank {
5   class CheckingsAccount playedBy Account {
6    callin withFee(float amount) {
7     ···
8     result = base.withFee(amount * FEE);
9     ···
10    return result;
11   }
12   withFee(float) ← replace withdraw(float)
13  }
14 }
15 ···
16 Bank bigBank; Account acc;
17 ···
18 bigBank.activate();
19 acc.withdraw(100.00);
```

**Figure 1: ObjectTeams/Java code to declare a role with additive behavior adaptations for accounts in a bank and its usage.**

represent more than a single specific class over time [3]. A similar observation was made in the domain of conceptual modeling [32].

The difference to the object-oriented paradigm is to classify each entity in the domain to either be of *natural type*, which is rigid and independent, or of *role type*, which is anti-rigid and dependent [25]. This dependency of role types is the foundation of the relation, which defines which natural type *fills* a role type. On the level of instances, a natural that plays a role in a context is extended with the behavior and properties of the role. Thus, roles allow separating the structure and relations of entities in a domain and the (context-dependent) behavioral adaptations [26, 40]. This change in behavior enables adaptive software by design and in consequence unanticipated adaptation [42].

## 2.2 Contextual Roles and the Semantic Gap

At runtime, instances of natural types are represented as objects with a compound type consisting of the natural type and a list of role types [23]. This dynamic extension happens orthogonal to the inheritance hierarchy of the natural type. Method lookup on these compound objects may return different call targets on objects with the same natural type but different role-playing relations. Role-oriented semantics often must be emulated, which in turn incurs a high runtime overhead [35]. The reason is the gap between the object model of these concepts and the object model of the underlying system or virtual machine (VM), a phenomenon already known from aspect-oriented programming [16]. In fact, heuristics often do not work with these dynamic extensions. As a consequence less optimized code compiled is produced from runtimes resulting in inferior performance [30, 35]. Implementation techniques range from interfaces and design patterns [4, 11, 41] over embedded DSLs [8, 13, 22, 27, 31, 39, 42, 44] to standalone programming languages [1, 18, 29]. The many implementations result from an inconsistent view on what features constitute a role-oriented programming language – forming a family of role languages [26].

ObjectTeams (OT) [17] is a programming model, which provides most of the features attributed to roles. It combines principles and features of context-oriented programming and aspect-oriented programming providing and allows class-wide as well as instance-local adaptations. The reference implementation ObjectTeams/Java (OT/J) extends Java featuring (unanticipated) adaptation [18]. To the best of our knowledge, it is the fastest implementation of contextual roles [35].

Fig. 1 shows a snippet of ObjectTeams/Java code.[1] Contexts are represented as `team class` who encapsulate their roles. A role declaration uses the `playedBy` statement to declare the *base type* (i.e., natural type), which is eligible to play the role. A role may define additional behavior in role functions. How a role interacts with its base type is defined by *bindings* (see Fig. 1 line 12). Bindings declare the method of the base type that is adapted, how the adaptation must be applied, i.e., before, after, or replacing the original method, and the role method (i.e., `callin`) going to be called. In terms used in aspect-oriented programming, a `callin` intercepts a method call and `callNext` proceeds the intercepted call.

The compiler assumes closed-world on the types of teams and roles that it type checks. It asserts that declarations of bindings have compatible type signatures to the declared base methods. For each binding the compiler generates lookup code that implements the dispatch to the declared role functions (e.g., `callReplace`), which is part of the program executed at runtime. The lookup code contains all possible dispatches to role functions defined inside a team class. We observed that since the advance of contextual roles all implementations rely on implementations based on patterns of delegation.

For classes referenced by bindings, i.e., base types, the assumption is open-world. At run-time there may be sub-classes loaded, which were not known at compile time. To realize such a mixed setting the compiler deduces type information, which is stored in the class' attributes. The OT/J language runtime adapts loaded classes and generates entry points into role dispatch to preserve the semantics.

Fig. 2 shows the evaluation from a function call of a program from Fig. 1. This scheme has been coined *recursive chaining wrapper*, as role method dispatch is implemented recursively over active team instances (i.e., `callNext`). The generated code and the recursive evaluation prohibit optimizations of the VM. For ObjectTeams/Java, this causes a performance penalty of 59.9× compared to a pure object-oriented design pattern implementation [35].

## 2.3 High-Performance Dynamic Language Runtimes and Partial Evaluation

The primary research vehicle we use to demonstrate our optimizations is Espresso, a meta-circular Java bytecode interpreter for the GraalVM [46]. The idea of a meta-circular virtual machine (VM) is not new and already has been explored with the Jikes RVM [2] and

[1]A detailed description of the language features is presented in [19].

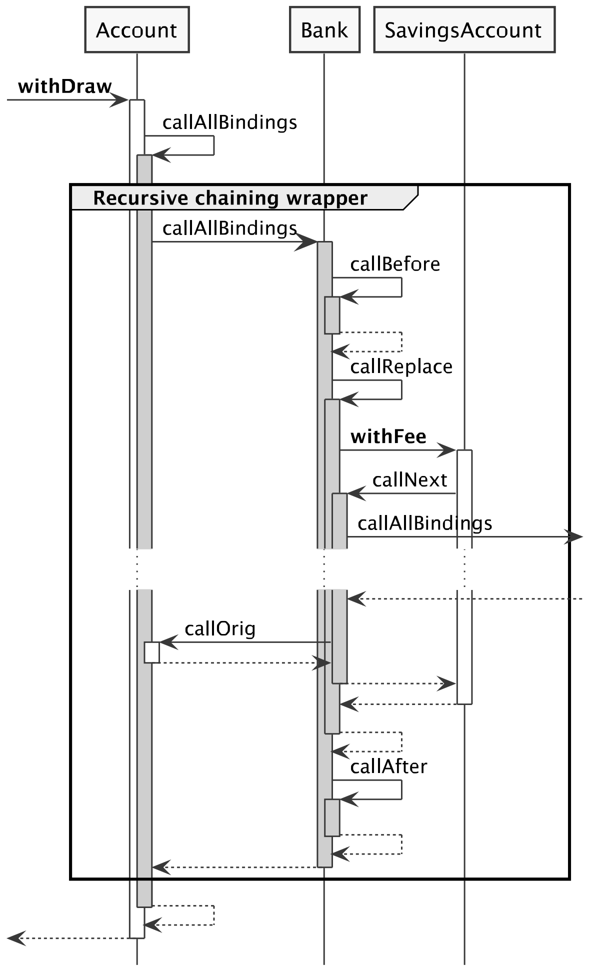

**Figure 2: Control flow of a role method dispatch. Grey boxes represent execution of framework code while white boxes represents behavior implementations.**

Maxine VM [43]. However, recent advances in VM optimization research re-opened the door for these kinds of VMs.

VMs follow different approaches to Just-in-Time (JIT) compilation. The Graal [9, 10] JIT compiler uses a method-based compilation model where the decision to optimize depends on heuristics such as method execution counts. Approaches such as PyPy [6], however, focus on compiling execution traces of the interpreter (meta-tracing) evaluating the program under execution. The work proposed in this paper builds on approaches using the former model of compilation.

Some compilers emit bytecode, a sequential intermediate representation (IR), which is the input to the dynamic language runtimes. One representative is Java Bytecode that is the input to a Java VM. During interpretation the language runtime may choose to *quicken* bytecodes, effectively replacing bytecodes with more specific ones using the feedback gathered during interpretation. Interpreters may employ more language-specific quickening-based optimizations [7].

A high-performance dynamic language runtime requires a custom VM and compiler increasing implementation and maintenance efforts. To overcome this burden Truffle [45] defines a DSL and execution model to implement self-modifying interpreters of abstract syntax trees (AST). During interpretation the type feedback of the interpreter is used to rewrite (i.e., specialize) the AST resulting in specialized code to execute. The compiler IR is generated after partial evaluating the specialized interpreter code, also called first Futamura projection [12]. The resulting code is further optimized and compiled by the Graal compiler to high-performance machine code.

## 3 RELATED WORK

This section introduces related work that focuses on improving the lookup and dispatch of context-oriented [20] and aspect-oriented programming languages [24].

Since object-oriented execution environments, i.e., virtual machines, do not understand aspect semantics, the aspect compiler produces a verbose description of aspects as some mechanisms cannot be directly represented in an object-oriented paradigm, which incurs a high overhead [16]. To close that gap Steamloom [14, 15] extends the Jikes RVM with aspect-oriented primitives to support dynamic aspects. This means to reduce the number of residuals at join point shadows and to provide facilities to efficiently execute advices. Method bodies affected by advice invocation are rewritten with aspect invocations. The required information is stored in an Aspect Instance Table (AIT), a runtime data structure embedded in the memory representation of classes. Advice invocation byte codes indexed aspects from the AIT to be invoked. However, Steamloom only implemented *before* and *after* advices. They remark that *around* advices (the pendant to *replace* in role-oriented programming) are a more demanding challenge [14].

The performance overhead of context-oriented programming language implementations has been attributed to the violation of assumptions a common language implementation holds about lookup. ContextPyPy [30] optimized the dispatch of layered methods in the meta-tracing JIT compiler PyPy by *promoting* context-oriented dispatches. Promotion makes the JIT compiler to ensure that traces are specialized regardless of whether the use of heuristics would result in a specialization or not. A preliminary implementation study dispatched context-oriented methods with `invokedynamic` and concludes that it benefits the performance of dispatching layered methods.

Polymorphic dispatch plans [36, 37] have been proposed as a solution to overcome the inherent overhead of role dispatch. ObjectTeams/InvokeDynamic (OT/Inv) [38] is the reference implementation extending ObjectTeams/Java using runtime generated dispatch graphs based on *invokedynamic* [34] byte codes. The approach supports the open-world assumption of lazily loaded types at run-time. Inspired by *partial evaluation*, the lookup uses runtime feedback to specialize role dispatch for observed values. To reduce the overhead of role dispatch the graph must only include calls to role methods without unnecessary delegations, e.g., bridge methods such as `callReplace` as seen in Fig. 2. The resulting graph can be optimized and reused in subsequent calls at the same call site. A *guard* ensures that invalid lookup results will not be executed. The

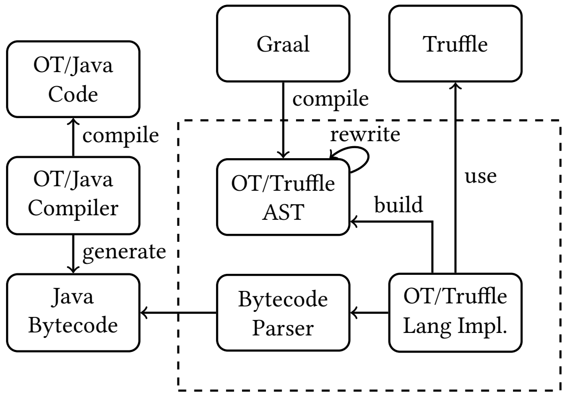

**Figure 3: Toplevel Architecture of ObjectTeams/Truffle.**

guard captures types of active contexts on initial invocation and evaluates on re-execution whether the active context is structurally equal. Otherwise, a new graph must be computed and appended to the call site, forming a polymorphic inline cache (PIC) [21]. The approach provides a speedup on role dispatch and is able to mitigate a fixed amount of variability. However, we observed that dispatch plans are a too coarse-grained and rigid structure requiring recomputation as contexts change. A megamorphic call site triggers cache contention, leading to slowdowns that can easily be of an order of magnitude due to constant recomputation of the whole graph.

## 4 VIRTUAL MACHINE ARCHITECTURE FOR CONTEXTUAL ROLES

We present ObjectTeams/Truffle, a VM implementation for contextual roles based on a model of contextual roles [18]. 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. This section first introduces the necessary ingredients a VM needs to support contextual roles. Then we demonstrate how these ingredients can be implemented in a meta-circular Java Bytecode interpreter written in Java.

### 4.1 Fundamentals of Virtual Machine Support for Roles

As presented in Sec. 3 the representation of contextual approaches and the dispatch to contextual methods impose a challenge for heuristics and JIT compilers in dynamic language runtimes. To support contextual roles the virtual machine must provide role-specific primitives and be able to derive optimizations over these primitives. In the following we will introduce key primitives that must be provided to enable VM support for contextual roles.

*4.1.1 Join Point Shadows.* The programming model of contextual roles divides monolithic entities into smaller, context-dependent components. In consequence it must be ensured that all relevant points in the application are found and observed, i.e., points in the program that must be woven. In aspect-oriented programming these points of the application are called *join point shadows*. 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 [5] to which all eligible call sites delegate to. The weaving will take place inside the envelope method. To support roles, a VM must provide support for join point shadows and facilities to weave locations in the program code.

*4.1.2 Plays Relation.* An object may simultaneously play roles in multiple contexts. The activation and deactivation of said contexts would change the order in which the roles are filled, i.e., in which their behavior influences the object itself. This relationship is represented by the *plays* relation, which maps an object to its roles [25]. Virtual machine support for roles would entail facilities to efficiently retrieve and store elements from that relation.

*4.1.3 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. The reasons are manyfold. First, it depends on the way contextual roles are represented, which often is reduced to role classes encapsulated by their context class. Second, the plays relation is retrieved from the context objects and is not stored with the role-playing object itself. This adds an extra level of indirection. Third, join points are evaluated at run-time. Repeated evaluation of the same join point should be omitted if the values of relevant properties do not change.

### 4.2 Extending Espresso with Support for Contextual Roles

In the following we will discuss how our prototypical implementation realizes the aforementioned fundamental requirements to provide support for contextual roles. A top-level diagram picturing the components of our approach in context is depicted in Fig. 3.

To handle join points we extended the ObjectTeams/Java compiler to annotate envelope methods. The compiler stores information for bindings in the attributes section of team classes. We extended the Java Bytecode parser to read the non-standard attributes and to store them in VM class representations. After the VM parses the Java Bytecodes the ObjectTeams/Truffle (OT/Truffle) language implementation, an extension of Espresso, will build the corresponding AST. We diverged from the original Espresso implementation whenever we encounter a reference to an envelope method. Call sites that reference annotated methods will be quickened into intrinsic AST nodes that execute role dispatch. The goal is to represent the ObjectTeams runtime model directly in VM data structures instead of using a language level meta-object protocol (MOP) and chains of delegation.

A fundamental part of the plays relation is the retrieval of roles played in the current context. In the model of ObjectTeams this is called *lifting* [19, §2.3]. To lift the base object, we represent the access to the plays relation 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 can be shared among multiple identical liftings that occur during a dispatch.

ObjectTeams/Truffle supports all the different types of role methods, i.e., before, after, and replace. Dispatch to roles is realized with multiple AST nodes where there is a node for each type of dispatch.

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

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

**Figure 4: The Java interfaces of the envelope methods `callAllBindings` and `callNext` in ObjectTeams/Java.**

The VM infers from the attributes the amount of role methods contributed by a respective context and the precedence of the role methods. This enables the generation of a concrete sequence of instructions instead of using loops or recursion in the compiled code of these nodes. 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 run-time.

In essence, the semantic gap is reduced due to the partial evaluation friendly design of the AST, the annotation of *compile-time* static values, the annotation of parameters to be cached, and the guards to decide when to reuse a value of a parameter. The result of the partial evaluation is a smaller IR with more opportunities for optimization by the Graal JIT compiler.

## 5 A SELF-MODIFYING AST TO REPRESENT CONTEXTUAL ROLE DISPATCH

This section discusses design decisions and implementation details of our proposed virtual machine support for contextual roles. We present how support for contextual roles can be captured in an AST effectively representing the language semantics for consumption by the VM, closing the semantic gap. The proposed structure is designed to be friendly to partial evaluation, a prerequisite to produce high-performance machine code.

### 5.1 Envelopes and Quickening

ObjectTeams/Java uses the envelope approach [5] to execute advices from join point shadows. The envelopes are the entry point to structured role dispatch. To initialize role dispatch OT/J defines the envelope *callAllBindings*. Its task is to collect the relevant runtime values that will be used in the dispatch logic to find the call targets. The signature, shown in Fig. 4, declares the required values. The weaver originally 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. In our implementation the envelope is replaced with a role dispatch AST node. Therefore, the Bytecode Parser reads the base method from the call stack when a call to the envelope is parsed for the first time before quickening the call node. The parameter `boundMethodId` is used as a predicate to identify the original method later.

To *proceed* (see `base.withFee(*)` in Fig. 1) from a replace callin OTJ provides the envelope *callNext*. 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. The solution is to exploit the fact that method activation records can be accessed as first-class entities at VM level. A method activation is normally recorded as a stack frame. The VM operates a stack of frames carrying information on the method whose execution has triggered its creation. In order to re-constitute the origin of the role call we placed a *cookie* on the stack that identifies the base method. Stack walking is able to capture the cookie.

An object-oriented VM treats the calls to the envelopes as calls to regular methods. Thus, the envelopes can be found in the sequence diagram shown in Fig. 2. To overcome the semantic gap, our VM replaces each envelope with an intrinsic method. Each intrinsic method quickens the `invokevirtual` bytecode node to the respective AST nodes for the envelopes *callAllBindings* and *callNext* encoding the role dispatch semantics. This is the entry point to the self-modifying AST.

### 5.2 Role Dispatch Optimization Opportunities

We know that efficient dispatch is key for performance. It has been shown that it is possible to optimize the implementation of contextual role dispatch by representing the dispatch with primitives that can be understood and optimized by a language runtime. For example, dispatch plans realized with the invokedynamic bytecode close the semantic gap and enable the generation of optimized code [38]. However, we observed that dispatch plans are a too coarse-grained and rigid structure requiring recomputation as contexts change.

As highlighted in Sec. 4 ObjectTeams/Truffle provides VM implementations for key primitives of contextual roles. Using an AST to represent role dispatch opens the opportunity to define specialized variants for important program states. For example, if there is no active context the role dispatch will not be executed at all but the original method. 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.

Our implementation exploits the fact that at each point at runtime the active contexts and their provided roles are known. An example AST is depicted in Fig. 5 representing the most important nodes to dispatch the statement `acc.withdraw(*)` taken from Fig. 1. Arrows do represent parent-child relations from the AST. Black arrows and nodes of the AST will contribute to the resulting generated code while grayed, dashed arrows and nodes are removed during partial evaluation because they are dominated by invalidated assumptions.

When executing the dispatch the AST node first is specialized w.r.t. the type of the first active context instance (see node *call all bindings* in Fig. 5). 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 polymorphic inline cache (PIC) [21] for contexts. This specialization forwards to a node that implements the dispatch to all provided role methods of a specific

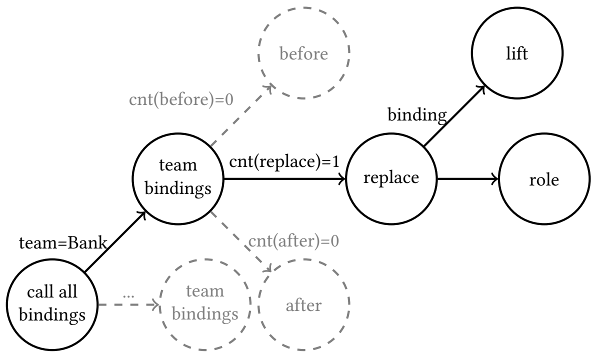

**Figure 5: The intrinsic AST that replaces the envelope *callAllBindings* created for `acc.withdraw(*)` in Fig. 1. Gray, dashed elements represent nodes that will not be compiled.**

context (node *team bindings*). When this node is created the VM reads the internally stored attributes of contexts and their bindings and switches the assumptions about the contributed role methods accordingly. For example, the `Bank` from Fig. 1 only provides a replace callin, which invalidates the assumptions representing the existence of before and after callins. The design of the AST allows to produce and optimize on a fine-grained granularity.

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. There exists an AST node for each type of callin to account for their different ascribed semantics. To reduce the amount of AST nodes we grouped the execution of each kind of callin in the respective nodes. This allows us to unroll the calls to role methods for before and after producing a sequence of instructions.

## 6 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 [18] and ObjectTeams with Dispatch Plans [38].

### 6.1 Benchmark Characterization

We used a typical synthetic benchmark we designed to compare different language implementations of the role-oriented concept [35]. The benchmark uses many demanding role-oriented programming features such as multiple active contexts, deep roles (i.e., roles playing roles), and multiple callins that are not easily built with object-oriented design patterns.

The benchmark describes a banking scenario. Persons and accounts are classes implementing basic behavior. For example, accounts can withdraw and deposit money. A bank is a compartment (i.e., context) where persons can play the role of customers. Accounts play roles that change the account's behavior such as different fees involved in withdrawing money from a checking account.

```java
bank.activate();
for (Account from :
  bank.getSavingAccounts()) {
  for (Account to :
    bank.getCheckingAccounts()) {
    Transaction transaction =
      new Transaction();
    transaction.activate();
    transaction.execute(from,to,amount);
    transaction.deactivate();
  }
}
bank.deactivate();
```

**Figure 6: The measured portion of the Bank benchmark dynamic case written in Object Teams/Java. `Bank` and `Transaction` are teams whose roles influence the `Account` behavior.**

We evaluate the approaches with a *dynamic* case with variable context activations to explore different characteristics of context-dependent software. Figure 6 shows the measured portion of the dynamic case. The inner-most loop models transactions as teams, which are activated and deactivated in every iteration. The activation and deactivation of the contained roles has no observable effect on the runtime until a relevant base method is called. The accounts play multiple roles (i.e., deep roles). For instance, the account that plays the roles of the source (`from`) in the transaction also plays the role of a `SavingsAccount` in the `Bank`. The target (`to`) of cash flow inside the transactions plays at the same time the role of a `CheckingsAccount` inside the bank. The activation and deactivation of the `Transaction` team changes the active roles for each of the role-playing instances.

### 6.2 Methodology

To gain valuable results the benchmark is executed for each approach with a set of different problem sizes. Every benchmark has been run 5 times for each data point. We measured the execution time per run and report the geometric mean and standard deviation for each problem size and approach. To observe whether there are scalability problems, we measured with different problem sizes. For the dynamic case we varied the problem size from 1.0 to 2.5 million transactions. The number of accounts participating in the transactions is increased proportional.

The benchmark was executed by the benchmark execution framework ReBench [28]. 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. The JDK our VM build is based on is LabsJDK CE17 JVMCI v23.0b01.

### 6.3 Performance Analysis

The results of the dynamic case are presented in Fig. 7. For comparison the runtime of the reference implementation ObjectTeams/Java [18], ObjectTeams/InvokeDynamic [38], and the approach proposed in this paper, ObjectTeams/Truffle are measured. It depicts

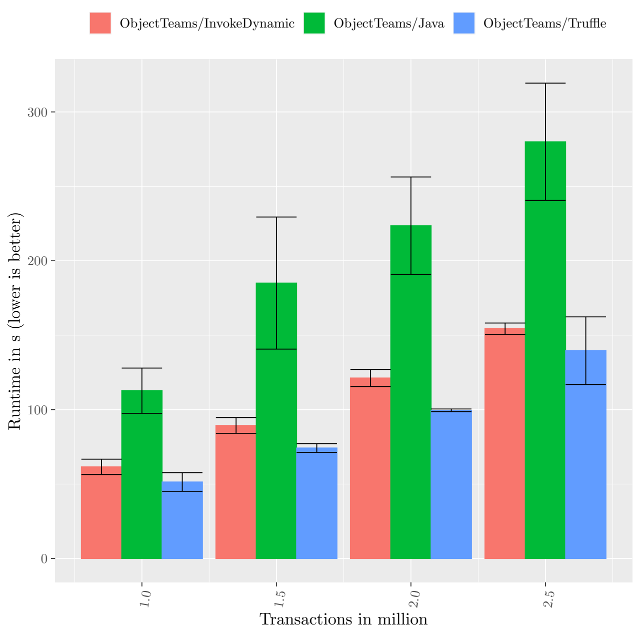

**Figure 7: Runtime in s for the dynamic case. The error bar shows the standard deviation of the run-time.**

the geometric mean of the measured runtimes for each approach and each problem size.

For ObjectTeams/Java and ObjectTeams/Truffle it is noticeable that the standard deviation increases with increasing problem size. The reason is that with these approaches garbage collection is executed more often increasing the execution time. One of the reasons may be the handling of runtime values used to execute the dispatch. The standard deviation for ObjectTeams/InvokeDynamic is lower across all problem sizes because it implements a copy-on-write optimization for the data structure representing the active team instances. We did not implement such an optimization for ObjectTeams/Truffle.

It stands out that ObjectTeams/Java is the slowest implementation in this benchmark. ObjectTeams/Truffle shows a speedup of up to 2.49× (mean 2.23×) compared to ObjectTeams/Java. Compared to ObjectTeams/InvokeDynamic our approach is up to 1.22× faster (mean 1.18×).

## 6.4 Threats to Validity

We are aware that Espresso is still an early prototype of a metacircular Java VM. It passes the Java Compatibility Kit but might have unequally good implementations for different parts of the Java Virtual Machine Specification. This could skew the results and improperly favor one implementation over the other.

The reference implementation ObjectTeams/Truffle supports a limited set of features provided by ObjectTeams/Java. It is possible that 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.

## 7 CONCLUSION AND FUTURE WORK

Context-dependent software continues to become increasingly important. The role concept is a promising candidate to build context-dependent software as contexts and behavioral adaptations can be directly represented in the language. This enables a flexible software development process suitable to build context-dependent software. In general, however, contextual role language implementations, as well as related aspect-oriented and context-oriented implementations suffer from a high runtime overhead when executing dispatches.

Since object-oriented virtual machines 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. This paper proposed prerequisites to support roles in a virtual machine. We present a 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 our approach achieves a speedup of up to 1.22×.

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 work. We see optimization potential for instances of roles, layers, and aspects that are *shared* and possess no private state.

In the current implementation the role-playing relation is still using the language level meta-object-protocol of ObjectTeams/Java. However, the heap and object graph are immediately accessible from within the virtual machine. In future work we want to extend the memory model of the VM and VM classes to manage the role-playing relation inside the virtual machine. This requires a VM internal handling for context activation and deactivation, which would make the language level meta-object-protocol of ObjectTeams/Java obsolete. For the performance characteristics of the implementation we predict that this would greatly improve speed and resource consumption.

## ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation within the Research Training Group Role-based Software Infrastructures for continuous-context-sensitive Systems (GRK 1907) and the Center for Advancing Electronics Dresden (cfaed). We want to thank Alfonso Peterssen and Christian Humer who provided detailed information about the architecture of Espresso and Truffle.

## REFERENCES

[1] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. 1995. Fibonacci: A Programming Language for Object Databases. *The VLDB Journal* 4, 3 (July 1995), 403–444.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal* 44, 2 (2005), 399–417.

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

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

[5] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. 2005. Envelope-Based Weaving for Faster Aspect Compilers. In *NODe 2005 GSEM 2005*, Vol. P-69. 3–18.

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. 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*. ACM Press, Genova, Italy, 18–25.

[7] Stefan Brunthaler. 2010. Efficient Interpretation Using Quickening. *ACM SIGPLAN Notices* 45, 12 (Dec. 2010), 1.

[8] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. 2004. A Role Model and Its Metaclass Implementation. *Information Systems* 29, 3 (May 2004), 235–270.

[9] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.

[10] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler. ACM Press, 187–193.

[11] Martin Fowler. 1997. Dealing with Roles. In *Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP 97)*.

[12] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.

[13] Georg Gottlob, Michael Schrefl, and Brigitte Röck. 1996. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems* 14, 3 (July 1996), 268–296.

[14] Michael Haupt and Mira Mezini. 2005. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet* 11, 3 (2005).

[15] Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. 2005. An Execution Layer for Aspect-Oriented Programming Languages. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. ACM Press, 142.

[16] Michael Haupt and Hans Schippers. 2007. A Machine Model for Aspect-Oriented Programming. In *ECOOP 2007 – Object-Oriented Programming*, Vol. 4609. Springer Berlin Heidelberg, Berlin, Heidelberg, 501–524.

[17] Stephan Herrmann. 2003. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World*. Vol. 2591. Springer Berlin Heidelberg, Berlin, Heidelberg, 248–264.

[18] Stephan Herrmann. 2007. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology* 2, 2 (2007), 181–207.

[19] Stephan Herrmann, Christine Hundt, and Marco Mosconi. 2011. OT/J Language Definition v1.3.

[20] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *The Journal of Object Technology* 7, 3 (2008), 125.

[21] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91 European Conference on Object-Oriented Programming*. Vol. 512. Springer-Verlag, Berlin/Heidelberg, 21–38.

[22] Bo Nørregaard Jørgensen and Eddy Truyen. 2003. Evolution of Collective Object Behavior in Presence of Simultaneous Client-Specific Views. In *Object-Oriented Information Systems*. Vol. 2817. Springer Berlin Heidelberg, Berlin, Heidelberg, 18–32.

[23] Tetsuo Kamina and Tetsuo Tamai. 2010. A Smooth Combination of Role-based Language and Context Activation. In *FOAL 2010 Proceedings*.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *ECOOP'97 — Object-Oriented Programming*. Vol. 1241. Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[25] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. 2015. A Combined Formal Model for Relational Context-Dependent Roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. Pittsburgh, PA, USA, 113–124.

[26] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*. Vol. 8706. Springer International Publishing, Cham, 141–160.

[27] Max Leuthäuser. 2017. Pure Embedding of Evolving Objects. In *The Ninth International Conference on Advanced Cognitive Technologies and Applications*. 22–30.

[28] Stefan Marr. 2018. ReBench: Execute and Document Benchmarks Reproducibly. (Aug. 2018).

[29] Supasit Monpratarnchai and Tamai Tetsuo. 2008. The Implementation and Execution Framework of a Role Model Based Language, EpsilonJ. IEEE, 269–276.

[30] Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2016. Optimizing Sideways Composition: Fast Context-oriented Programming in ContextPyPy. ACM Press, 13–20.

[31] Michael Pradel and Martin Odersky. 2008. SCALA ROLES A Lightweight Approach towards Reusable Collaborations. In *ICSOFT 2008 - Proceedings of the 3rd International Conference on Software and Data Technologies*. 13–20.

[32] Trygve Reenskaug, Per Wold, and Odd Arilc Lehne. 1996. *Working with Objects: The OOram Software Engineering Method*. Manning, Greenwich.

[33] Dirk Riehle and Thomas Gross. 1998. 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, 117–133.

[34] John R. Rose. 2009. Bytecodes Meet Combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*. ACM Press, Orlando, Florida, 1–11.

[35] Lars Schütze and Jeronimo Castrillon. 2017. 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*. ACM Press, Brussels, Belgium, 1–6.

[36] Lars Schütze and Jeronimo Castrillon. 2019. Efficient Late Binding of Dynamic Function Compositions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2019*. ACM Press, Athens, Greece, 141–151.

[37] Lars Schütze and Jeronimo Castrillon. 2020. Efficient Dispatch of Multi-object Polymorphic Call Sites in Contextual Role-Oriented Programming Languages. In *17th International Conference on Managed Programming Languages and Runtimes*. ACM, Virtual UK, 52–62.

[38] Lars Schütze, Cornelius Kummer, and Jeronimo Castrillon. 2022. 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)*. ACM, Berlin Germany, 27–34.

[39] Yannis Smaragdakis and Don Batory. 2002. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology* 11, 2 (April 2002), 215–255.

[40] Friedrich Steimann. 2000. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data & Knowledge Engineering* 35, 1 (Oct. 2000), 83–106.

[41] Friedrich Steimann. 2001. Role= Interface: A Merger of Concepts. *Journal of Object-Oriented Programming* (2001).

[42] Nguonly Taing, Markus Wutzler, Thomas Springer, Nicolás Cardozo, and Alexander Schill. 2016. Consistent Unanticipated Adaptation for Context-Dependent Applications. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*. ACM, Rome Italy, 33–38.

[43] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization* 9, 4 (Jan. 2013), 1–24.

[44] R.K. Wong and H.L. Chau. 1998. Method Dispatching and Type Safety for Objects with Multiple Roles. In *Proceedings. Technology of Object-Oriented Languages and Systems, TOOLS 25 (Cat. No.97TB100239)*. IEEE Comput. Soc, Melbourne, Vic., Australia, 286–296.

[45] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. ACM Press, 662–676.

[46] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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*. ACM Press, Indianapolis, Indiana, USA, 187–204.

Received 2023-05-22; accepted 2023-06-15