Kent Academic Repository

Full text document (pdf)

Citation for published version

Chari, Guido and Garbervetsky, Diego and Marr, Stefan and Ducasse, Stéphane (2018) Fully Reflective Execution Environments: Virtual Machines for More Flexible Software. IEEE Transactions on Software Engineering . ISSN 0098-5589.

DOI

https://doi.org/10.1109/TSE.2018.2812715

Link to record in KAR

http://kar.kent.ac.uk/66604/

Document Version

Author's Accepted Manuscript

Copyright & reuse

Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research

The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries

For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR
Kent Academic Repository

University of Kent

Fully Reflective Execution Environments

Virtual Machines for More Flexible Software

Guido Chari, Diego Garbervetsky, Stefan Marr, and Stéphane Ducasse

Abstract—VMs are complex pieces of software that implement programming language semantics in an efficient, portable, and secure way. Unfortunately, mainstream VMs provide applications with few mechanisms to alter execution semantics or memory management at run time. We argue that this limits the evolvability and maintainability of running systems for both, the application domain, *e.g.*, to support unforeseen requirements, and the VM domain, *e.g.*, to modify the organization of objects in memory.
This work explores the idea of incorporating reflective capabilities into the VM domain and analyzes its impact in the context of software adaptation tasks. We characterize the notion of a fully reflective VM, a kind of VM that provides means for its own observability and modifiability at run time. This enables programming languages to adapt the underlying VM to changing requirements. We propose a reference architecture for such VMs and present TruffleMATE as a prototype for this architecture. We evaluate the mechanisms TruffleMATE provides to deal with unanticipated dynamic adaptation scenarios for security, optimization, and profiling aspects. In contrast to existing alternatives, we observe that TruffleMATE is able to handle all scenarios, using less than 50 lines of code for each, and without interfering with the application's logic.

Index Terms—Reflection, Virtual machines, Metaobject protocols, Dynamic adaptation

1 Introduction

Most software systems evolve during their lifetime [1]. In many cases, the required modifications must be performed without interrupting their execution [2]. For this kind of applications, the boundary between development-time and run-time blurs [3]. Consequently, it is beneficial to extend the run-time adaptability from the applications also to the system that executes them, *i.e.*, the virtual machine itself.

Managed languages such as Java, Python, and JavaScript run on top of highly complex software artifacts known as virtual machines (VMs). VMs perform various tasks such as implementing the language's semantics, providing dynamic compilation, adaptive optimizations, automatic memory management and enforcing application's security. To cope with demanding performance requirements, the majority of today's industrial-strength VMs are written in low-level languages such as C and C++. After compilation, these VMs limit the ability of applications to observe and adapt them at run time. Even research VMs do not enable significant observability and interactivity with themselves at run-time [4], [5], [6], [7].

Because VMs execute application code, they must be aware of the inner aspects of application entities such as objects, methods, and statements. But this relation is unidirectional: applications have only limited control over how the VM manages them. In general, applications are designed to be unaware of the existence of the VM. We advocate that a new generation of VMs should promote a *bidirectional* communication between themselves and the applications they execute. Our hypothesis is that *opening* VMs to applications makes software more flexible and adaptable.[1]

To test our hypothesis, we propose the idea of fully reflective virtual machines: VMs exposing their whole structure and behavior to the applications at run time. A fully reflective VM allows developers to observe and adapt the VM *on-the-fly* enabling modifications from simple adaptations to fine-grained tuning of applications. This provides application developers with novel capabilities to adapt running applications. In addition to modifying the application logic, with a fully reflective VM, developers can adapt the application by modifying the behavior of the VM supporting its execution.

As an example for the potential of such VMs, consider a server application that has to run without interruption. In case a security issue is found, one might want to use a custom security analysis to determine its impact with respect to application and user data. Since the system is live, for safety reasons the analysis must not modify any data, *i.e.*, the security analysis must be side-effects free.

In this paper, we show that a fully reflective VM provides the mechanisms to enforce the read-only guarantee of the aforementioned analysis, as well as support for other adaptation scenarios, at the language level in a simple and unanticipated manner (cf. Section 7). We start by discussing the main characteristics of a fully reflective VM and we suggest a reference architecture. The architecture defines a metaobject protocol (MOP) [9] to operate programmatically on the structure and behavior of the VM. The MOP ensures that VM and application communicate in a controlled fashion via a predefined API. The architecture is implemented in TruffleMATE, a reflective VM implemented on top of the Truffle framework [10]. TruffleMATE provides extensive reflective capabilities at the VM level.

- *Chari and Garbervetsky are with the Departamento de Computación, FCEyN, UBA, and ICC CONICET, Argentina. E-mail: {gchari,diegog}@dc.uba.ar*
- *Marr is with the University of Kent, United Kingdom. E-mail: s.marr@kent.ac.uk*
- *Ducasse - RMoD project team, Inria Lille - Nord Europe, France. E-mail: stephane.ducasse@inria.fr*

1. This hypothesis was partially outlined in a previous paper [8] that this journal version extends.

To validate our hypothesis and assess TruffleMATE's novel capabilities we analyze how it handles unanticipated adaptation scenarios. These scenarios require adaptations to both, an application and its VM. Our experiments provide empirical evidence about the feasibility and usefulness of a fully reflective VM as an appealing alternative for supporting on-the-fly software adaptation. While not the focus of this paper, we also present preliminary indications suggesting that reflective VMs can run efficiently.

The contributions of this paper are:

- The proposal of a new approach for building VMs promoting a bidirectional communication between a VM and applications running on top.
- A reference architecture for building reflective VMs featuring a MOP for handling VM-level reflection from the application level.
- TruffleMATE,[2] a reflective virtual machine implemented using the Truffle framework and supporting the Smalltalk programming language.
- An empirical validation with case studies on one of the most challenging instances of software adaptation: handling unanticipated adaptation scenarios *on-the-fly*. The results show that our approach can handle unanticipated adaptations in a reasonably simple and homogeneous manner compared to other approaches.

2 Motivation

In this section we motivate the need for fully reflective VMs. We start by describing an adaptation case study that is then used throughout the rest of the paper. Then we discuss different aspects of why existing approaches are not suitable to properly handle each scenario.

2.1 Running Example

Unanticipated adaptations include all possible changes to a system that were not identified at design time. This paper focuses on applying such changes at run time. As motivation, let us consider an application that needs to be keep running to avoid affecting customers. However, at some point customers complain about long response times. Consequently, a developer profiles the application to understand what is going on. She finds that a likely root cause is high memory usage of the application. Thus, she wants to reduce memory consumption and improve the overall system performance.

From this example, we distill the following adaptations scenarios:

Read-only Protection. Immutability in software refers to the ability to turn parts of the program state accessible only for reading. Immutability has been proven useful for software development & testing, optimizations, and verification among other tasks [11], [12]. In OO languages, immutability has been applied at the object level, *i.e.*, for protecting individual objects, or at the reference level, *i.e.*, for protecting any object (transitively) accessed through a particular reference.

2. https://github.com/charig/truffleMate/tree/papers/TSE2017

In our example, the analysis scripts must not modify any data to protect its integrity. That is, data should be only accessible for reading. To achieve this goal without interrupting the whole system, the references from the analysis scripts to the application must have a read-only behavior.

Application Profiling. Application profiling has been shown to be useful for software development tasks such as program comprehension [13], debugging [14], and optimizing [15]. These tasks typically benefit from calling context information, for instance, to improve the speed and correctness of software maintenance tasks [16] or decide whether a method requires further optimization [17]. In our example, in addition to tracking activated methods and the context of their activations, the profiled information should also include method arguments, return values, and accesses to local variables.

Efficient Field Access. Most VMs organize their objects in memory as a continuous block of cells. Using this organization, reading an object field implies that the whole cache-line including the field is loaded into the processor cache. Therefore, iterating over a large amount of objects and accessing only few of their fields, potentially spanning multiple cache-lines, can result in sub-optimal performance because of frequent cache misses.

It was identified that this is in part slowing down the example application. The application aggregates statistics for a large number of objects based on a single field. Compared to realizing the same behavior using an optimized relational database, this code is suboptimal since the involved objects are large. Hence, the object memory representation should be improved to increase performance.

Efficient Field Storage. Dynamic languages usually enable the addition and removal of object fields at run time. To optimize the representation of such a dynamic structure of unknown size and shape, the notions of *maps* [18] and object *shapes* [19] have been proposed. Essentially, they keep track and cache object structures and field types at run time. Exploiting this information using speculative type specializations enables a record-like representation in memory where field accesses can be mapped to direct memory accesses.

Due to the nature of the application, many of its classes have a large number of fields to represent customer data. However, many fields for a large number of objects are never used. Since the language is class-based, and the shape of these classes keep track of all the fields, memory usage is suboptimal. Therefore, the data representation should be more compact to free up memory.

2.2 Direct and Indirect Adaptations

Our running example includes requirements similar to adaptations discussed in literature [12], [20], [16], [21], [22]. Generally, unanticipated scenarios involve adaptations of low-level structures such as the object representation in memory as well as behavioral features, *e.g.*, to profile execution. Furthermore, they may require the adaption of fine-grained entities including individual objects.

We identified two different ways for approaching these adaptation requirements:

- **Direct adaptation** is the redefinition of the semantics of exactly the required operations for the necessary scope. As such, there is an *ontological correspondence* between the requirement and the adaptation.
- **Indirect adaptation** is the redefinition of semantics by wrapping around (intercepting) the required operations and redirecting the execution flow. When the adaptation scope exceeds the requirement, it is also considered to be indirect.

From our experience [23], indirect adaptation leads to a number of issues: 1) the set of operations intercepted is usually an over-approximation of the points in the program that need adaptation; 2) maintainability is reduced because any application change can require a corresponding update to any interception point; 3) debugging is cumbersome because the application's methods could become polluted with instrumentation features; 4) composing adaptations can require complex run-time tests at each interception point reducing performance; 5) if an operation is not interceptable (*i.e.*, built-in operations) it can not be adapted.

To illustrate direct adaptations, below we sketch how to approach the scenario from the running example in a direct way. In sections 7.2.2, 7.2.3, 7.3.1, and 7.3.2 we provide detailed implementations.

Immutable References. To handle the read-only protection requirement one option would be to make all objects immutable. However, this option is infeasible if the application was designed to perform computations by mutating objects. To avoid this issue, another option is to forbid mutations made only by the analysis script. This can be achieved by using read-only references [12], which enforce that objects accessed through them cannot be mutated. Unfortunately, dynamic languages rarely reify the concept of references.

Arnaud et al. proposed *Handles* as a reification of read-only references [20]. Handles are proxies to objects that delegate every operation to their targets but prevent mutation. Handles must be transparent, i.e., it should be impossible to distinguish whether an object is accessed directly or through a handle. Moreover, every object accessed through a handle is wrapped into another handle to ensure immutability through the complete chain of accessed objects.

Handles can be used for handling the read-only scenario in dynamic languages in a direct way but require VM support for customization of the method activation, method lookup, and read operations.

Profiling Using Calling Context Trees. An approach for capturing the context of the method activations, needed by the profiling requirement, is to represent the information using calling context trees (CCT), a data structure that compactly represents method executions grouped by calling contexts [24]. In a CCT, each tree node represents the execution of a method and its ancestor nodes represent its callers, *i.e.*, the methods that would be found in the call stack. A direct way of gathering this information is to redefine how the language activates methods (only for the required objects) so that before a method is activated the required information is logged in the corresponding CCT node.

Fast Aggregation with Columnar Objects. In relational databases, the overhead in data intensive applications is avoided by organizing the data in columns instead of rows [25], [26]. A direct adaptation approach for the scenario requiring fast field access is to organize the fields (of the class denoting the relational table) in a columnar manner, so that they are stored in subsequent memory cells [21].

Efficient Representation of Sparse Objects. Shapes are responsible of the mapping between each field and its actual position in memory. Consequently, a direct adaptation approach at run time to reduce the memory consumption caused by unused fields is to introduce shapes that only assign space for used fields.

2.3 Approaching the Adaptive Scenarios

Reflection in programming languages is a mechanism for programs to express computations about themselves, enabling the observation (*introspection*) and/or modification (*intercession*) of their *structure* and *behavior* [27].

Reflective systems are a suitable approach for adapting software directly as they enable an interaction with program elements without resorting to intermediaries. This is in contrast to alternatives such as aspect-oriented programming [28] and context-oriented programming [29], which usually enable only indirect adaptations (cf. section 9.3).

We now analyze the reflective capabilities that state-of-the art solutions promote to deal with these scenarios.

2.3.1 Contemporary Reflective Systems

Conceptually, reflective systems are able to approach any adaptive scenario directly. However, even the approaches with most extensive reflective capabilities found in literature suffer from a common limitation: their reifications cover only a limited subset of low-level entities [30], [31]. As a consequence, they fail to handle direct adaptations requiring changes to VM-related aspects such as those suggested for our four scenarios.

2.3.2 Reflective Capabilities in Virtual Machines

A common characteristic of VMs is that they are made of several intertwined components each coping with complex responsibilities [32]. However, performance is usually the most critical aspect for VM developers, because a VM's performance affects the throughput of the programs it executes. Consequently, mainstream VMs are usually designed prioritizing performance rather than other aspects such as modularity or adaptability. This makes them complex artifacts, difficult to observe and adapt at run time. For example, most mainstream VMs do not allow to experiment with an alternative algorithm for the *method lookup* mechanism or to dynamically introduce immutability for a predetermined set of objects. Instead, such changes require developers to implement the new behavior, recompile the VM, and restart the application.

Some research projects do however provide interaction mechanisms at run time (cf. section 9). For example, Pinocchio [33] is, to the best of our knowledge, the most complete solution in terms of reflective capabilities. Pinocchio is a first-class interpreter extensible from the language level implementing Smith's tower of interpreters [27]. Since it provides reflective capabilities only for the operational semantics of the language (interpreter), Pinocchio does not support the direct approaches suggested for the optimization

scenarios as it cannot adapt structural entities such as the layout of objects.

2.4 Problem Statement and Hypothesis of Work

Existing reflective systems cannot properly handle fine-grained unanticipated adaptations in a direct fashion. Especially when these adaptations involve VM concepts such as object layouts, operational semantics, etc.

Based on these observations, we hypothesize that *to handle unanticipated adaptations directly at the language-level, a VM should provide abstractions to mold the semantics of the whole system at the application and the VM level*.

3 Background

This section introduces concepts required throughout the paper.

3.1 Execution Environments

We define an *Execution Environment* (EE) as a set of software components in charge of executing programs written in a specific programming language. For instance, an EE for an object-oriented (OO) language is responsible for executing statements, managing how objects are represented in memory, and collecting the objects that are no longer used. EEs are also known as VMs or managed runtimes. Hereinafter, we refer to them mainly as VMs.

3.2 Reflective Architectures

A programming language has a reflective architecture if it provides tools for reflective computations [34]. For instance, reflective architectures for object-oriented programming languages usually rely on metaobjects [35] that describe the structure and behavior of language-level concepts such as objects and methods. The set of metaobjects that represents a particular object constitutes the object's meta level. The metaobjects describing all base-level objects in an application constitute the application's meta-level. In reflective languages and architectures, metaobjects and their corresponding base-level objects must be *causally connected*: changes in metaobjects must lead to a corresponding effect upon their associated base-level objects [34].

Metaobject protocols (MOPs) [9] are APIs that give users the ability to modify the language's behavior and implementation by using metaobjects. They are an elegant solution for handling non-functional aspects of applications [36], [37], [38]. Iguana/J [30], object-centric debugging [39], and Albedo [40] adopt MOPs to deal with low-level concerns.

To improve distribution, deployment, and general purpose metaprogramming of reflective architectures, Bracha and Ungar [41] proposed the *Mirror* design principles: i) *Encapsulation:* MOPs should not expose implementation details ii) *Stratification:* MOPs should enforce a separation between the application behavior and the reflective code iii) *Ontological correspondence:* the meta-level reifications[3] must map one-to-one to concepts of the base-level domain.

3. To reify: to model as a first class entity.

3.2.1 Reflective Dimensions

To the best of our knowledge, there is no metric for assessing the degree of reflective capabilities supported by a system. Thus, we propose the following reflective dimensions:

- *Domain-breadth* measures both, how many entities of a domain (inter-entity) are reified, or how many features are included in the reification of each entity (intra-entity).
- *Domain-depth* measures the number of meta-levels reified for each entity.

For instance, when considering the reification of features from classes such as instance or class variables, methods, and inheritance relationships, we measure intra-entity domain-breadth. On the other hand, domain-depth considers the levels of *metaclasses* available for a programmatic interaction. For instance, in a language such as Smalltalk, developers can inspect the metalevel of the metalevel, *e.g.*, the metaclass of a metaclass. The term *full reflection* refers to the complete reflective coverage of both the domain-breadth and domain-depth dimensions of reflection.

3.2.2 Application-level vs. VM-level Reflection

Commonly, reflective computations are distinguished based on whether they are used for introspection or intercession, as well as whether they affect behavioral or structural elements. However, this distinction does not consider abstraction levels. For example, adding fields to an object and modifying its memory location are both characterized as structural intercession, even though both operations deal with different levels of abstraction. The first refers to object fields, an application-level concept, while the latter refers to memory (VM level). It is common in reflective languages, such as Smalltalk and JavaScript, to support the addition of fields at run time, without providing support for customizing the memory locations of fields. Since the distinction is relevant for our work, we introduce the following categories:

- *Application-level reflection* refers to metaprograms that work with objects, classes, methods, or object fields of the application's domain model.
- *VM-level reflection* refers to metaprograms that regard operational semantics, execution stack, method lookup, memory management, or any reification of low-level aspects handled by the VM.

3.2.3 Reflective Challenges

To enable the customization of any VM-level behavior an *intercession handler* is added to delegate its execution to the corresponding language-level entity (the metalevel) where it can be customized [42]. This handler enforces the causal connection, but has a performance cost. Furthermore, whenever a shift to the metalevel occurs the VM must take care of translating its data representations and those of the language back and forth. As a consequence, completeness and performance are in tension: incorporating more reflection into a system, *i.e.*, making it more complete, increases the flexibility at the cost of affecting its performance.

When designing a reflective system, this tension must be resolved. Fortunately, just-in-time compilers can significantly reduce the overhead of intercession handlers whenever the

usage of the reflective capabilities are moderate or exhibit stable usage patterns [43], [44].

4 Fully Reflective Execution Environments

This section defines three maxims that a VM must follow to be considered a fully reflective execution environment and proposes a reference architecture.

4.1 Main Characteristics

Maxim 1. **Universal Reflective Capabilities**: *VMs must provide intercession and introspection capabilities for every entity at both, the application and the VM level.*

VMs usually impose a rigid boundary with the applications. This is beneficial in terms of security, portability, and performance, but restricts the possibility of the applications to affect the VM behavior at run time. To overcome this limitation, we advocate for VMs exposing reflective capabilities at the VM level.

Maxim 2. **Uniform Reflective Abstractions**: *VMs must provide the same language tools for interacting with both, the application and the VM levels.*

Ideally, developers that work in different domains should be able to focus on a single tool for dealing with reflective computations at different levels. Since uniform abstractions help to improve the understandability and evolvability of the programming environment [45], [46], we argue that VM-level reflection must use the same application-level mechanisms to avoid increasing the complexity. For instance, if the language provides reflection via a MOP, VM-level reflection must also be supported by a MOP.

Maxim 3. **Separation of Application and VM**: *An application should not need to be designed explicitly to support observability and adaptability. Instead, the VM should provide the necessary capabilities.*

To separate concerns, an application must focus on the problem domain, while orthogonal concerns should be handled separately. For example, similar to aspect-oriented programming, a cross-cutting low-level adaptation such as logging must not affect the application's domain model. Hence, it is important that the abstraction for dealing with reflection enables a clear separation between the application and the VM domains. However, this is rarely the case and concerns such as logging or more efficient data representations have to be realized at the application level.

4.2 Mate: A Reference Architecture

Figure 1 presents Mate, the high-level architecture we propose for fully reflective VMs. It is divided into the application and the VM layers with arrows representing different kinds of interactions between components. To represent a wide range of object-oriented languages, Mate relies only on the notions of objects and methods as its core elements.

The bottom layer comprises only essential VM-level entities for executing expressions, realizing objects, and managing memory in OO languages. Further refinements may extend those, for instance, by incorporating entities such

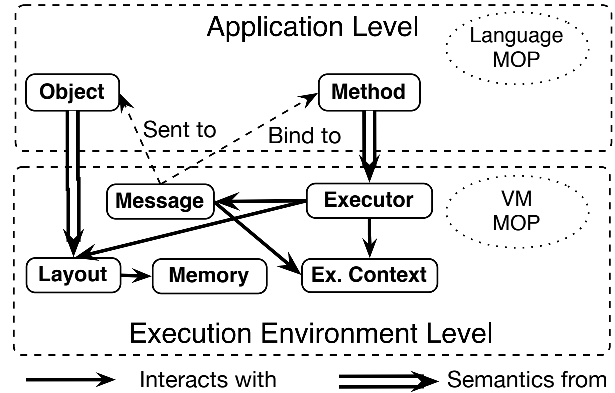

Fig. 1: High-Level reference architecture of a fully reflective environment for an object-oriented generic programming language.

as threads, I/O, and execution traces. To realize universal reflection (maxim 1), all of these entities must provide reflective capabilities, possibly using a MOP that is complementary to an application-level MOP. This honors the uniformity required by maxim 2 and complies with the separation of domains required by maxim 3.

The main responsibilities of each VM-level entity are:

- `Executor` is responsible for interpreting and possibly optimizing methods. It defines the operational semantics of the language.
- `Execution context` manages the stack and the information that the executor uses for executing a method including the given receiver and arguments.
- `Message` is responsible for the binding of messages to methods (method lookup) and the corresponding method activation that creates the execution *context* in which the method will be *executed*.
- `Layouts` describe the concrete organization of the internal data of objects.
- `Memory` realizes in combination with `Layouts` the memory representation of objects. This includes defining read/write accesses as well as memory allocation and garbage collection.

5 A VM-Level MOP

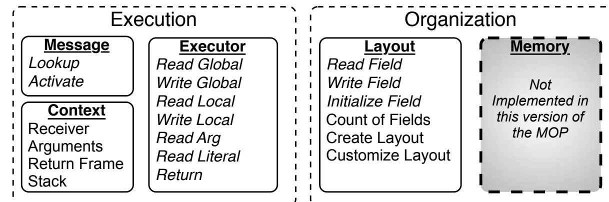

Fig. 2: Our Metaobject Protocol is designed to enable *unforeseen software adaptations* by providing access for inspecting and modifying the VM level. The operations highlighted in italics represent behavioral aspects, while the others represent structural aspects of the VM.

This section presents the design of a VM-level MOP for the Mate architecture. Instead of devising a design for all aspects of each component, we focus on a MOP capable of handling unanticipated dynamic adaptations such as those presented in section 2: read-only protection, profiling, performance optimizations, and space optimizations on-the-fly. Figure 2 presents a sketch of the resulting MOP. The metaclasses are grouped into two clusters: one for concepts of execution and the other for the organization of data. Compared to the Mate architecture, the resulting MOP only leaves out the memory metaclass, which we do not need for the adaptive scenarios (evaluated in section 7). The domain-breadth reflective capabilities per metaclass are:

- `Message`: Allows developers to specialize the method lookup algorithm and the method activation mechanism. In section 7.2.2 we show examples of its application for handling adaptation scenarios.
- `Executor`: Allows developers to redefine the behavior of *each* operation giving semantics to language constructs. For instance, in a bytecode-based implementation it allows the redefinition of each individual bytecode. The scenario of section 7.2.3 illustrates the usage of most of them.
- `Execution Context`: Makes it possible to observe the receiver, the arguments, the caller's context, and the stack values for each executing method. We show an example of its usage in section 7.2.2.
- `Layout`: Provides means to modify the behavior of operations interacting with object's fields. Specifically, the reading, writing, and initialization of fields. It also allows the introspection and intercession of the memory organization of objects. Usage examples can be found in section 7.3.

5.1 How to Use the MOP

There are two different ways of using the MOP depending on whether one wants to interact with behavioral or structural aspects. For structural aspects, such as number of object fields or the current values on the execution stack, reflection is handled by observing or altering the corresponding metaobject directly. For instance, a *layout* describing the structure of each object is accessible for observation and modification. Furthermore, an *execution context* is accessible for every method invocation describing the contextual values.

For customizing the behavioral operations of the MOP (highlighted using italic letters in figure 2), users should describe the new behaviors. To do so, the metaclasses of the MOP (`Message`, `Executor`, and `Layout`) can be subclassed to override the methods defining the corresponding language's behaviors. `ExecutionContext` is not included because its operations are accessors for structural aspects only.

Finally, the customized metaclasses must be attached to an auxiliary metaclass called `Environment`. Environments merely aggregate metaobjects into one single object, which minimizes memory usage, because each entity only needs a single pointer to an *environment* to customize its whole VM-level semantics.

Example. To illustrate how to specialize these metaclasses consider a simplification of the read-only protection scenario in section 2.1. Let us assume that the analyzed module is not used in the meantime. Thereby, we can avoid introducing read-only references and implement object immutability. To do that we only need to customize the *Write field* operation from `Executor` to make it throw an exception whenever the system tries to change the value of a field.

The left part of figure 3 shows a configuration of metaobjects implementing the scenario. For readability, the figure distinguishes between `metaclasses` and *metaobjects*. `Immutable`, which extends `Executor`, implements the read-only semantics by specializing the `writeField` method. To use it, an `environment` object is created (*aReadonlySemantics*) and the customization is attached to its Executor field. Now *aReadonlySemantics* can be attach to objects to make them immutable. This configuration is generated by:

1) Subclassing `Executor` with `Immutable` and specializing the `writeField` operation.
2) Instantiating `Immutable`. The metaobject is named *aImmutable*.
3) Instantiating `Environment`. The object is named *aReadonlySemantics*.
4) Assigning *aImmutable* to the Executor field of *aReadonlySemantics*.

The middle part of figure 3 shows a configuration to support the sparse objects scenario presented in 2.1. The goal is to reduce memory consumption by compressing objects containing uninitialized fields. This is done by using a customized layout that implements a dictionary-like object layout, using only space for fields that are used. To this end, `Compactable` customizes the semantics of *read field*, *write field*, and *field count* from `Layout`. Analogous to the immutable case, an instance of `Compactable` (*aCompactable*) is attached to an instance of `Environment` (*aCompactEnvironment*) but this time to the Layout field. Objects that want to use a sparse layout must then be linked to (*aCompactEnvironment*).

Finally, the right side of the figure shows how to combine customizations from different metaclasses in a single *Environment* (*aCompactReadonlyEnvironment*).

5.2 Characteristics of MATE's MOP

Below, we discuss the most relevant design characteristics of our MOP.

Composability/Modularity. MOPs adhering to the *ontological correspondence* principle isolate the reflective capabilities into separate objects that correspond to domain-level structures. This promotes composability [47]. Honoring the principle, each VM-level entity is a separate metaclass.

Scoping. The `Environment` metaclass links base-level entities with the VM metalevel. For fine-grained scoping, every base-level object has a link to an *environment* metaobject which describes how the VM must operate on itself. In addition, method activations can also link to an *environment* to redefine the semantics of all the operations executed within the method. Finally, *environments* can be set globally as well. To avoid ambiguity, the *environment* applied to a more specific context precedes the others.

Activation/deactivation. The explicit separation between the VM-level and application-level MOPs comply with the Mirror's principle of *stratification*. Bracha and Ungar claim

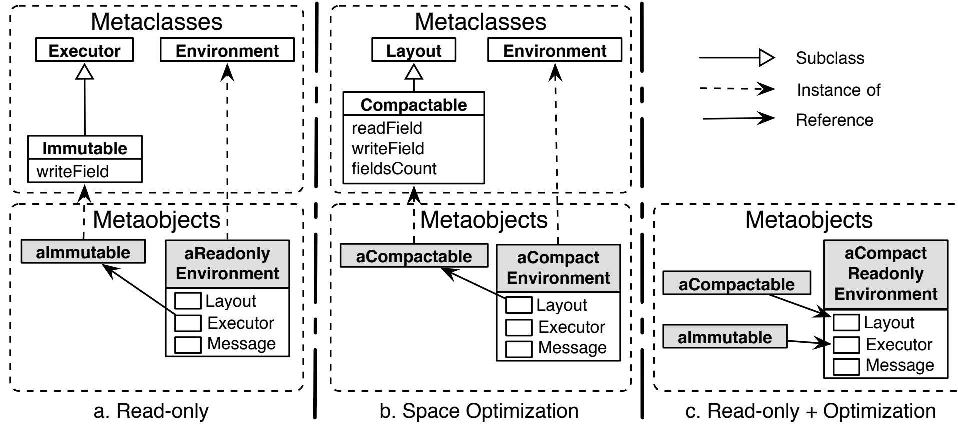

Fig. 3: Configuration of metaobjects for the adaptation scenarios. In the left (a.) there is a configuration of metaobjects for realizing the read-only scenario. In the middle (b) for the space optimization scenario. The right-hand of the figure shows how to combine the previous configurations in an *environment* for realizing both, read-only and space optimization adaptations at the same time.

that adhering to this principle helps to avoid overheads when VM-level reflection is not needed [41]. In our case, for activating/deactivating VM-level behavior redefinitions, it is only needed to add/remove the *environment* metaobject from the corresponding base-level entity.

5.3 VM-level Behavioral Reflection

To realize the causal connection for behavioral reflection, Mate uses *intercession handling* [42]. Concretely, before executing any VM-level operation included in the MOP such as invoking a method, or accessing a local variable, the VM tests whether there is a metaobject redefining it. If not, the standard VM-level operation executes. In case the operation is redefined, the VM delegates the responsibility to the corresponding language-level method. The following algorithm defines the concrete intercession handling for our MOP:

```
1  def IH(frame, operation) {
2    result = NOMETAOBJECT;
3    if (level is Meta) return result;
4    metaobject = getReceiver().getMetaobject();
5    if (metaobject != NOMETAOBJECT)
6      result = metaobject.activateFor(operation);
7    if (metaobject == NOMETAOBJECT or result == null) {
8      metaobject = frame.getMetaobject();
9      if (metaobject != NOMETAOBJECT)
10       result = metaobject.activateFor(operation);
11   }
12   if (metaobject == NOMETAOBJECT or result == null) {
13     metaobject = getGlobalMetaobject(operation);
14     if (metaobject != null)
15       result = metaobject.activateFor(operation);
16   }
17   return result;
18 }
```

Listing 1: Algorithm describing the intercession handling process.

The first two lines of the algorithm show how the metaregression problem is solved by Mate. Metaregression is an endless recursion caused by a metaobject calling a base-level behavior that is also redefined by the same metaobject [48]. A general solution to this problem is making reflective architectures *context-aware* by reifying the execution level [49]. This provides an extra parameter for defining the semantics of the operations. We use a simplification of *meta-contexts* by providing only two different levels of execution: *meta* and *base*. Every time the VM delegates the execution to the metalevel, the depth level is set to *meta* and no further delegations are possible until the operation returns.

Starting with line 4, the intercession handling checks for the existence of a metaobject associated with the *subject* of the current VM operation (*e.g.*, in a variable read operation, the concrete object owning the variable). In case there is one, the intercession handling activates the language-level redefinition of the current operation in the metaobject. In case there is none, or it does not redefine the VM operation being executed (returns null), the intercession handling looks for a metaobject associated with the current *frame* of execution. Again, if there is no metaobject or it does not redefined the operation, the intercession handling checks for a global metaobject redefining the current operation. If the operation is not redefined at any level, the intercession handling returns the `NOMETAOBJECT` constant and the VM executes the default behavior.

Completeness. The domain-depth completeness of the behavioral part of our MOP is limited to two levels, meta and base, as a simple solution to avoid metaregression. The main reason for this solution is that none of the considered adaptation scenarios required a higher domain-depth. The domain-breadth completeness was already discussed for each metaclass of the MOP in the introduction to this section.

Note that our approach does not support adding new intercession handling points at run time, *i.e.*, the behavioral reflective capabilities of the VM cannot be increased on-the-fly. This means the MOP is fixed at compile time.

5.3.1 VM-level Structural Reflection

We reify the structure of base-level entities using the fields of metaobjects. To guarantee the causal connection, the behavior of base-level entities is defined by the corresponding

metaobject. Consequently, this mechanism enables a program to observe and change the value of metaobject fields with instantaneous effects on the base-level entity.

Completeness. Analogous to the behavioral case, structural reflective capabilities of the MOP cannot be increased at run time. Concretely, no new reification of structural VM level entities, for instance the trace of method activations, can be realized after compiling the VM. Considering the domain-depth dimension, we faced a metaregression issue, analogous to the one discussed for intercession handlers, but with layouts. Since layout metaobjects are also first-class objects their layout is defined by another layout metaobject. Since our adaptation scenarios do not require further capabilities, we provide non-redefinable layouts for metaobjects limiting the domain-depth to two levels.

6 TRUFFLEMATE

In previous work [8], we presented a Smalltalk interpreter to investigate a subset of the ideas discussed in this paper. The interpreter was a naive prototype which suffered from performance overheads of 2 to 3 orders of magnitude in comparison to an optimized VM. To perform a more comprehensive validation, we developed TruffleMATE, a reflective VM implemented on top of the Truffle language implementation framework.

TruffleMATE allows us to validate our MOP for more realistic scenarios. However, building an industrial-strength VM is a highly resource demanding task and not our aim. Instead, our goal is to provide a platform to validate the fundamental aspects behind reflective VMs. This section presents TruffleMATE and the required technical background on Truffle and the Graal compiler.

6.1 Truffle and Graal

The Truffle framework allows developers to build programming languages by expressing their semantics as abstract syntax trees (ASTs). To realize self-optimizing AST interpreters [10], Truffle comes with a DSL to specify specializations. Specializations express execution patterns for the cases when a more optimized version of an operation can be used. For instance, performing a simple addition when both arguments are integers instead of relying on a general method that would handle arbitrary types for each argument.

In combination with the Graal just-in-time (JIT) compiler [50], languages implemented in Truffle can reach the peak performance of today's industrial-strength VMs [51], [52]. For instance, Graal.js, a JavaScript on top of Truffle reaches on average the same peak performance as V8 [53], [52] for a specific set of benchmarks. Truffle languages reach an average performance of about 2x slower than Java on HotSpot. Furthermore, Truffle-based implementations of Ruby and R outperform the best performing existing implementations, JRuby and GNU R correspondingly. To reach this performance, Truffle applies partial evaluation on the specialized ASTs. Graal compiles the result to native code applying optimizations such as inlining and escape analysis.

6.2 SOM and TruffleSOM

The Simple Object Machine (SOM) [54] is a Smalltalk implementation designed to avoid inessential complexity. It includes fundamental language concepts such as objects, classes, closures, and non-local returns. Following the Smalltalk tradition, control structures such as `if` or `while` are defined as polymorphic methods on objects and rely on closures and non-local returns for realizing their behavior. TruffleSOM [43], [51] is a SOM implementation using Truffle.

6.3 From TruffleSOM to TruffleMATE

TruffleMATE extends TruffleSOM with two goals: 1) turn it into a reflective VM and 2) become a full-fledged Smalltalk VM. For the first, it implements the VM-level MOP presented in section 5 completely. For the latter, we include support for additional primitive type optimizations, literal arrays, cascade message sends, support for file access, exceptions, and Smalltalk's streams [46]. As a result, TruffleMate is able to run programs developed for two compatible open-source Smalltalk implementations, Squeak [55] and Pharo [56], provided they do not use graphical user interfaces or concurrency. Below, we provide a brief overview of the most significant differences between TruffleMATE and TruffleSOM.

Environments. In TruffleMATE, the semantics of each behavioral VM operation can be redefined at three different scoping levels: individual objects, the whole execution of a method, or globally (cf. section 5.2). To support the former, we add a field to every object referring to an *environment*. To support the second, we modified the calling convention so that every method receives an extra *implicit* parameter with an *environment*. This *environment* governs the semantics of all operations executed within this method no matter the subjects. For the global scope we introduced a global variable also referring to an *environment*.

Intercession Handling. We modified the VM's behavioral operations included in the MOP (indicated with italics in figure 2) so that they execute the corresponding intercession handling introduced in section 5.3. Recall that the intercession handling checks for the existence of a metaobject that applies to any of the possible scopes and delegate the execution to the corresponding application-level method or the standard VM operation.

Objects. TruffleSOM provides its own object model. To enable custom object representations using the MOP, we provide a new object model based on the Truffle Object Storage Model (OSM) [19] and expose layouts to the application (shapes in OSM jargon) via primitive operations. This enables to reflectively inspect, create, and customize layouts of individual objects at run time. In addition, TruffleMATE incorporates support for basic Smalltalk object types such as characters and bytes.

Execution Stack. In Smalltalk, the *execution context* (frame) of the current method is reflectively accessible via the `thisContext` keyword. Concretely, Smalltalk natively implements the `Context` metaclass of the MOP. TruffleSOM does not support this behavior. We support it in TruffleMATE.

7 Evaluation of TruffleMATE's Adaptation Capabilities

This section assesses how TruffleMATE handles *unanticipated fine-grained adaptations* at *run time*. For each scenario from section 2.1 we discuss a direct adaptation using TruffleMATE and compare it with existing alternatives. Furthermore, we detail how TruffleMATE subsumes partial behavioral reflection [42], which we consider to be the most complete adaptation framework at the language level. We conclude with a basic performance evaluation of our prototype.[4]

7.1 Evaluation Criteria

For the evaluation, we compare a set of quality attributes between TruffleMATE and the best language-level solution judging our reflective VM as either better, the same, or worse. The attributes are:

- *Succinctness*: the number of lines of code (#LOCs) needed for the adaptation. This includes dependencies on software artifacts such as instrumentation frameworks. It does not include the code that is part of the runtime itself. Consequently, TruffleMATE's intercession handling code is not included. Nevertheless, its overhead is less than 20 LOC.
- *Forward compatibility*: whether an adaptation persists during the evolution of the application. For instance, if the adaptation monitors some entities and a new piece of code is added, it is desirable that the new code automatically includes the monitoring behavior.
- *Scoping*: whether the approach provides explicit ways to apply changes at different levels of granularity.
- *Modularity*: indicates whether it is possible to implement the adaptation without polluting the application's logic.

7.2 Extending Language Features

This section describes direct adaptations in TruffleMATE for the first two scenarios requiring new language features to be added to the application at run time: object and reference immutability as well as low-level profiling capabilities.

7.2.1 Per-Object Immutability

Recall the read-only scenario from section 2.1. It requires the implementation of reference immutability on the fly to protect the system against unintended modifications. We start by describing and analyzing a direct way of implementing classic per-object immutability in TruffleMATE. In the following section, we extend this adaptation to achieve the required reference immutability semantics.

Object Immutability in TruffleMATE. As the following code snippet shows, we simply install a metaobject in the target object to redefine the write operation:

```
1  class Immutable extends Layout {
2    def writeField(aNumber, anObject) {
3      throw new InvalidWriteException();
4    }
5  }
6
7  immutableLayout = new Immutable();
8  immutableEnvironment = new Environment();
9  immutableEnvironment.setLayout(immutableLayout);
10 obj = new Object();
11 obj.setEnvironment(immutableEnvironment);
```

On lines 1-5, we subclass `Layout` and overload the *writeField* operation to signal an exception instead of changing the field. From line 6 on, the code creates the *environment* and links the immutable layout metaobject to it. The last line installs the *environment* in a new object. To deactivate immutability, we can simply unset the environment: `obj.setEnvironment(NOMETAOBJECT)`.

Comparison to other approaches. Zibin et al.'s [11] approach enforce immutability by relying on static typing. It requires modifications of the application-level code to include type annotations and recompilation to recheck the annotations. In TruffleMATE object immutability can instead be applied to a dynamically-typed environment. Therefore, it is more *succinct* since it does not require modifications of the type system. It is also more *modular* since instead of using type annotations within method bodies, the adaptation logic is isolated in metaobjects. There is no significant difference in any of the other criteria.

An alternative for implementing object immutability in dynamic environments is to instrument every method in the system that may eventually modify the state of immutable objects. This has negative impact on *succinctness*, *activation*, and *forward compatibility* compared to TruffleMATE. One way to mitigate these issues is to change the granularity of the immutability property from objects to classes. This means that all instances of a class are mutable or immutable. However, this might be prohibitively restrictive. It makes the alternative worse than TruffleMATE in terms of *scoping* without resolving the problems in the other criteria. In the next section, we discuss an alternative, immutability based on dynamic proxies.

Finally, VisualWorks Smalltalk[5] and some Ruby versions use a mutability flag in each instance to support per-object immutability. Every time an object is to be changed, the VM first checks this flag and raises an error if mutation is forbidden. These solutions do not suffer from the aforementioned limitations and should be better than TruffleMATE in terms of *activation impact* since the adaptation is *hardwired* at the VM level. On the other hand, they require dedicated VM support and thus cannot be considered as solutions for an unanticipated adaptation.

7.2.2 Reference Immutability

To provide reference immutability, we extend TruffleMATE at run time with Arnaud's *handles* [20]. Recall from Section 2.2 that handles are like proxies to objects that delegate every operation to their targets, except for mutating operations. Handles must be transparent: a user should not be able to distinguish whether an object is accessed directly or through a handle. Moreover, any object accessed through a handle is wrapped into another handle, propagating immutability through the chain of accesses from a handle.

Reference Immutability in TruffleMATE. The code below implements handles using our MOP:

4. Instructions for reproducing all the experiments can be found at http://github.com/charig/truffleMate/tree/papers/TSE2017.

5. http://www.cincomsmalltalk.com

```
1   class ImmutableMessage extends Message (
2     def lookup(subject, aMethodName) {
3       return super.lookup(subject.getTarget(), aMethodName);
4     }
5
6     def activateWithArgs(subject, aMethod, args) {
7       if (aMethod.name().equals("=="))
8         args["receiver"] = subject.getTarget();
9     }
10  )
11
12  class ImmutableLayout extends Layout (
13    def read(subject, anIndex) {
14      return new Handle(subject.instanceVarAt(anIndex));
15    }
16
17    def write(subject, anIndex, aValue) {
18      InvalidWriteException signal
19    }
20  }
21
22  class Handle extends Object = (
23    fields: target;
24    static fields: semantics;
25    semantics = new Environment(
26      new ImmutableSemantics(),
27      new ImmutableLayout());
28
29    Constructor Handle(anObject) {
30      target = anObject;
31      this.installEnvironment(Handle.getSemantics());
32    }
33
34    def getTarget() {
35      return target;
36    }
37
38    def static getSemantics() {
39      return semantics;
40    }
41  )
42
43  class Object = (
44    def readonly() {
45      return Handle(this);
46    }
47  )
```

Our implementation encapsulates the semantics of the immutability, the transparency, and the propagation properties of handles in four methods within two metaclasses: `ImmutableMessage` and `ImmutableLayout`. Below, the description for each of these properties:

- Immutability: We reuse the *write* method from the previous example, which signals an exception.
- Propagation: We redefine the read operation (Lines 13-15) to enforce that every access to a field of an object referenced by a handle returns a handle wrapping the corresponding field value. In addition, in Line 3 we delegate the lookup to the superclass but customize the first parameter. This ensures that the method is looked up in the class of the original subject (the target) and not in the handle. In combination, these two methods ensure that messages sent to a handle execute the method from the target and that side-effects are disabled in the chain of activations.
- Transparency: The activation ensures that the identity of read-only references is preserved by overloading the receiver when activating the equality test in Lines 7-8. Therefore, handles are transparent and all operations appear to be performed directly on the target object.

Comparison to other dynamic approaches. Arnaud's implementation of *handles* [20] duplicates classes. Every class that needs to support immutability has a corresponding *shadow class*. Shadow classes wrap all methods that change state to forbid the modification. To maintain *forward compatibility*, this mechanism requires changes in the compiler to synchronize shadows every time a method of the original class changes. In addition, it has a significant *activation impact* since it requires to instrument the whole system. Furthermore, the approach requires to adapt the method lookup so that the transparency property can be enforced. More recently, an approach based on dynamic proxies relaxed the transparency guarantees and modeled handles without requiring modifications to the VM [57]. Nevertheless, maintaining these proxies requires code generation very similar to that for hidden classes. Consequently, *forward compatibility* and the *activation impact* are still costly.

We showed that in TruffleMATE both, per-object and per-reference immutability, can be activated at run time even if it was not anticipated. In contrast to some of the aforementioned approaches, ad-hoc support, such as shadow classes or method duplications, is not needed. Finally, the adaptations do not affect application logic and are transparent. For instance, they are not observable when application methods are debugged. Accordingly, *forward compatibility* is automatic, *i.e.*, eventual modifications to the application would not affect immutability because the adaptation semantics are encapsulated in the corresponding metaobjects.

7.2.3 Profiling Applications Using Calling Context Trees

As discussed in section 2.2, we want to adapt TruffleMATE to collect profiling information to identify performance issues. Specifically, we want to build a calling context tree collecting for each calling context the number of activations, activation arguments, return values, and local variable accesses.

Profiling in TruffleMATE. To construct a CCT, we need to gather contextual and low-level information at run time. This can be done with TruffleMATE's reflective capabilities:

```
1   CCT extends object (
2     static fields: instance; // Singleton
3     fields: root, current;
4
5     static def getInstance() {
6       return instance;
7     }
8
9     def logActivation(aMethod, arguments) {
10      /* Look or create a node hanging from current targeted
11      to aMethod. Then update current to the corresponding
12      node and log the arguments to current.*/
13    }
14  )
15
16  CCTActivation extends Message (
17    def activateWithArguments(aMethod, arguments) {
```

```
18          CCT.getInstance().logActivation(aMethod, arguments);
19          return aMethod.activateWithSemantics(this);
20      }
21  )
22
23  CCTOperations extends Executor (
24      def returnValue(aValue) {
25          if (thisContex.getMethod().isConstructor())
26              CCT.getInstance().returnCreatedObject(aValue);
27          else
28              CCT.getInstance().returnValue(aValue);
29          return aValue;
30      }
31  )
32
33  def main(args) {
34      profilingEnv = new Environment();
35      profilingEnv.setMessage(new CCTActivation());
36      profilingEnv.setExecutor(new CCTOperations());
37      thisContext.installEnvironment(profilingEnv);
38      // Below would follow the original main code
39  }
```

For developing the CCT data structure we followed the algorithm described in literature. We refrain from describing it here to focus on the usage of the MOP. To profile all the required information we introduce a single metaobject redefining only the method activation from the `Message` metaclass and the return operation from the `Executor`. Note that this metaobject was designed to work at a method activation granularity, *i.e.*, the semantics of whole method executions are governed by the corresponding *environment*.

Lines 1-14 sketches the essential adaptations of our CCT implementation. Lines 16-21 show how to redefine the method activation so that it informs the current CCT which method is activated and what the actual arguments are. To ensure the whole application is profiled, in line 19 the metaobject installs itself in the frame of the method to be activated. Lines 23-31 redefine the return operation so that the CCT receives the return value. In case the value is the result of an instantiation, the CCT records also the number of allocated bytes based on the type of the object.

Finally, lines 33-39 adapt the entry point of the application to activate the profiling semantics. We create an *environment* containing both aforementioned metaobjects and install it in the current activation frame, accessed in TruffleMATE with the *thisContext* keyword.

Comparison to other approaches. The profiling information can be gathered using aspect-oriented programming (AOP) or instrumentation frameworks [58], [59], [60], [61], [62], [63]. For instance Senseo provides information about running applications to IDEs and collects information similar to our requirements [16]. The information is also collected using CCTs and includes number of invocations along with receiver and argument types, return types, number of object allocations, and allocated bytes.

Senseo relies on Major [60], an AOP-based profiler. The advantages of this approach is its high accuracy, portability, flexibility, and extensibility. Recent AOP implementations even promise only moderate overheads. However to guarantee *forward compatibility*, the instrumentation and aspect weaving might need to be reapplied when new methods are added to the system. Depending on the specific approach, *succinctness* might be affected because of the dependency on a heavy-weight framework, and it might exhibit a high *activation impact*.

Using TruffleMATE, the same information can be gathered by combining two metaobjects (with a few lines of code each) that automatically propagate through the chain of activations during profiling. Furthermore, the evolution of the application does not affect the profiling behavior. Therefore, TruffleMATE does not suffer from the main drawbacks the other approaches suffer.

An alternative technique, *sampling profiling* [64], [65], [66], produces partial execution information. We refrain here from a comparison because we are interested in producing the same information as Senseo, which requires precise profiling.

7.3 Extending Data Representation

This section presents two case studies requiring structural adaptations of objects at run time. Consequently, they assess the structural reflective capabilities of the MOP.

7.3.1 Fast Aggregation with Columnar Objects

Recall from section 2.2 that organizing data in columns instead of rows [25], [26] can improve the performance of analytical algorithms.

Columnar Objects in TruffleMATE. We implement a columnar organization of object fields [21], at run time, using the MOP. The code below shows *ColumnarData*, the main auxiliary class we need to implement columnar classes in TruffleMATE. It stores each of the fields of the class in a separate array with one fixed position for each instance of the class. Then objects become *proxies* storing the object class and the column position. Mattis et al. [21] demonstrate that within loops traversing such collections, these proxies can be optimized out. An escape analysis can optimize them to the integers representing the column numbers.

```
1   class ColumnarData (
2       static fields: columnarInstanceEnv;
3       fields: columnarData, lastPosition;
4
5
6       Constructor ColumnarData(aClass, initialSize) = (
7           columnarData = new Array(aClass.instVars().length());
8           lastPosition = 0.
9           for (i = 1; columnarData.length(); i++) {
10              columnarData[i] = new Array(initialSize);
11          }
12      }
13
14      def getData(aProxy, anIndex) {
15          return columnarData[anIndex][aProxy.getIndex()];
16      }
17
18      def setData(aProxy, anIndex, aValue) {
19          columnarData[anIndex][aProxy.getIndex()] = aValue;
20      }
21
22      def newInstance() {
23          lastPosition = lastPosition + 1;
24          proxy = new ColumnarProxy(lastPosition);
25          proxy.installEnvironment(columnarInstanceEnv);
26          return proxy;
27      }
28  )
```

Our code above also defines the logic for accessing (Lines 13-15) and storing (Lines 17-19) object fields in the corresponding array and at the proper index. Accordingly, instances of a columnar class read and store to their corresponding index in the array representation.

ColumnarData is also responsible for creating the new instances of classes featuring a columnar representation (Lines 21-26). At object creation time, we select a new index for the class arrays storing the data and create a kind of proxy storing only this index. This proxy represents the new object. Similar to the handles in the immutability scenario, it is desirable for this in-memory organization to be transparent. To do so without changing the code of the getters and setters of the class, line 24 installs the following metaobject:

```
1  class ColumnarFieldSemantics extends LayoutMO (
2    static fields: columnarClasses;
3    columnarClasses = new Hash();
4
5    def read(anIndex) {
6      return columnarDataFor(this).getData(this, anIndex);
7    }
8
9    def write(anIndex, aValue) {
10     columnarDataFor(this).setData(this, anIndex, aValue);
11   }
12
13   def static columnarDataFor(aProxy) {
14     return ColumnarClasses.at(aProxy.class());
15   }
16 )
```

The *ColumnarFieldSemantics* metaobject defines a static association between each columnar class and its ColumnarData organization. Furthermore, it redefines the read and write operations so that they access the data in the ColumnarData arrays at the corresponding index.

Comparison to other approaches. Our approach only requires to install a metaobject redefining the reading and writing of fields. The same behavior could be achieved by removing the fields from the classes and changing the *getter/setter* for every field so they access their index in the data arrays. However, using this alternative direct accesses to fields must be forbidden, *i.e.*, by enforcing the usage of accessor methods. Furthermore, it is not *modular* because it modifies the application classes, and it has a significant *activation impact*. Using both, our approach and the alternative, *forward compatibility* is not provided automatically: if the object's layout changes, for instance because a field is added, the columnar adaptation must be updated too. However, since the objects already have a metaobject installed in TruffleMATE it is only required to create the columnar array for the new field. The alternative requires to adapt all its accesses in addition to creating the array.

Avoiding most of these problems, Mattis et al. [21] present a Python library, which introduces annotations for classes so that their fields are organized in a columnar layout. They show that tracing compilers are able to optimize operations such as selection, filtering, and mapping on large amounts of data if the objects are organized in a columnar layout in memory. Developers must annotate the classes that should use a columnar layout, but the algorithms and the code of the application for field accessing remains the same. The approach relies on implementing proxies ensuring the interception and redirection of every access to any columnar field. However, there remain limitations concerning object identity (transparency of proxies) without a dedicated VM.

7.3.2 Efficient Representation of Sparse Objects

In our running example we described the need to use a more efficient memory representation for sparse objects to avoid allocating memory that is not going to be used.

To make the scenario more concrete, let us consider a *Person* class with 20 fields for the attributes on an individual. Since Smalltalk uses a record-like representation for every instance, if most of the fields are not used, a significant amount of allocated memory remains unused.

Dictionary-based Layouts using TruffleMATE. To implement a dictionary-like representation in TruffleMATE and reduce the memory consumption at run time we can dynamically assign a layout to sparse objects that: 1) stores fewer fields per instance and 2) provide a metaobject for realizing a dictionary-like behavior for this layout.

For instance, consider a layout storing ten fields. Hence, Person's instances filling the individual information with less than five fields could use this layout. The reason why the layout has ten fields instead of five is that our dictionary representation needs two fields for representing each original field: the first stores the data and the second the index (or name) of the original field. Below, the customization of `Layout` for implementing the dictionary behavior:

```
1  class DictionaryBasedLayout extends Layout {
2    def read(anIndex) {
3      index = this.getIndexForField(anIndex);
4      if (index.isNull())
5        return null; // field has not been written
6      else
7        return this.instVarAt(index);
8    }
9
10   def writeField(anIndex, anObject) {
11     index = this.getIndexForField(anIndex);
12     if (index.isNull())
13       throw new NoMoreSpaceException();
14     else {
15       this.instVarAtPut(index, anObject);
16       this.instVarAtPut(index + 1, aNumber);
17     }
18   }
19
20   def fieldsCount() {
21     this.class().instanceVariables().size();
22   }
23 }
```

The `DictionaryBasedLayout` metaclass adapts the reading and writing of fields. For both operations we first obtain the index for the field, and then perform the operation. To ensure consistency and transparency, we also redefine the method that returns the number of fields of an object. This dictionary-based representation saves space when less than half of the fields are used.

Comparison to other approaches. Another approach to reduce memory usage could be to migrate sparse instances to new classes. This would however require to change both, the application code and the instantiation points, thus affecting *modularity*. Depending on the application, this may also require significant code changes and potentially the use

of a large DSU frameworks for migrating objects at run time [67]. These issues reduces *succinctness*. Furthermore, since this alternative requires adaptation of the target classes, it has a significant *activation/deactivation impact*. Finally, the adaptation is scattered through the application code, which reduces *modularity* and *forward compatibility*. In contrast, a reflective VM does not depend on how the application is implemented and does not suffer from the aforementioned drawbacks. The application code remains the same. It only requires to copy fields from one representation to the other for sparse objects.

Similar to MATE, Verwaest et al. [22] reify layouts at the language level. However, since the VM is not aware of them, these layouts can be bypassed by primitive operations that do not recognize those constructs. On the other hand, dynamic languages such as JavaScript or PHP represent properties of objects with hashed-based dictionaries. Without aggressive optimization, this is inefficient when most of the fields are used. Using optimizations such as maps or shapes [19], which do not allocate memory for unused fields, this is not an issue. Unfortunately, these solutions are provided at the VM level and are thus not an option for unanticipated adaptation.

In contrast to other VM-level solutions, with a reflective VM one can easily switch between dictionary-based and array-based representations at the granularity of objects. One could even replicate other shape-based approaches at the language level, which could further optimize memory usage.

7.4 Partial Behavioral Reflection with TruffleMate

Unanticipated Partial Behavioral Reflection (UPBR) [31] provides the most complete set of reflective capabilities for unanticipated adaptations that we are aware of. UPBR is an extension of partial behavioral reflection (PBR) [42], a language-level framework for realizing reflective computations efficiently. Reflex is the first tool implementing PBR and its main limitation is that it instruments Java bytecodes at load time, and thus, adaptations have to be anticipated. UPBR is an implementation for Smalltalk, based on run-time instrumentation, that supports the realization of adaptations at run time.

We now describe how TruffleMATE supports all capabilities of UPBR [42]. The reverse does however not hold. With UPBR, object layouts can not be adapted. Figure 4, taken from the original PBR paper, illustrates its main concepts and their relationships:

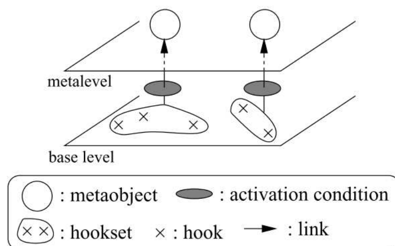

Fig. 4: Partial Behavioral Reflection main concepts. Taken from [42].

- *Hooksets* are sets of operation's execution points (hooks). For each adaptation, hooksets denote all execution points in the system that may need to delegate their execution to the metalevel.
- *Links* bind metaobjects with hooksets and establish the protocol between the base and metalevels. They specify the information passed to the metaobject and for instance the control given to the corresponding metaobject, *i.e.*, acting before, after, or around the intercepted operation.
- For each association between a link and hook, an activation condition is executed to test whether to delegate to the metalevel.

To illustrate these concepts let us consider an ad-hoc profiling scenario: we need to log the activation of every method with more than one argument. In this case the hookset would denote all method activations in the application. The link binds every operation of the hookset with a metaobject responsible of doing the actual logging. The activation condition checks whether the activated method contains more than one argument.

Hooksets, Links, and Conditions in TruffleMATE. The UPBR concepts can be modeled in a straightforward manner using the MOP:

- Hooksets are sets of execution points. The operations reified by `Executor` and `Message` metaclasses capture all of these points. Furthermore, TruffleMATE enables us to scope the selection of these operations to single instances.
- The link connects the base and meta level by binding hooksets to language-level methods defined in metaobjects. *Environment* metaobjects play the same role. In general, TruffleMATE supports an even finer-grained control of these bindings, because it allows us to redefine the language semantics even for a single operation and on a single object.
- Since in TruffleMATE the intercession handling's activation conditions are fixed, the link activation condition must be expressed inside the methods of the metaobject.

Comparing the approaches. We showed how TruffleMATE can model PBR concepts. The inverse however does not hold: PBR can not express the structural scenarios presented in the previous section. Moreover, PBR implementations presented in literature depend on instrumentation frameworks. This makes TruffleMATE more *succinct*. In cases such as the aforementioned profiling of method activations, PBR must instrument all methods of an application making the *activation/deactivation impact* significant. *Forward compatibility* is also more complex than with TruffleMATE, because PBR must reexecute the instrumentation after updating any method.

7.5 Performance

Until this point, the evaluation focused on our main goal: to show that reflective capabilities at the VM level are useful for handling unanticipated software adaptation. However, in many cases applications need to run efficiently to be usable.

A major concern about reflective VMs is that they may not run efficiently due to the overheads incurred by the intercession handling and the dispatching to language-level methods instead of realizing the VM behavior natively. In previous

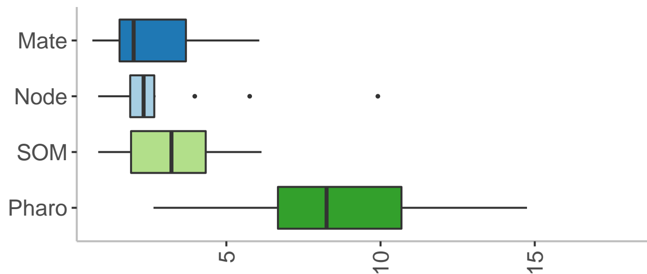

Fig. 5: Overhead factor of different dynamic programming language implementations normalized to Java. Benchmarks were selected from Marr et al. in [52]. The suite was designed for cross-comparing language implementations.

TABLE 1: Overall Baseline Results

Runtime	OF	CI-95%	Sd.	Min	Max	Median
Mate	2.64	±0.94	1.62	0.66	6.07	1.99
Node	2.95	±1.35	2.34	0.84	9.91	2.32
SOM	3.35	±1.03	1.78	0.84	6.14	3.22
Pharo	8.74	±2.3	3.80	2.64	14.75	8.25

work [44], we investigated optimizations and found preliminary indications that a reflective VMs like TruffleMATE can reach performance comparable with mainstream VMs. Specifically, for scenarios with low meta-level variability, we were able to combine speculative optimizations such as dispatch chains [43] and branch speculation to mitigate the overheads introduced by intercession handling.

To complement our qualitative evaluation and show indications that fully reflective VMs have the potential to be competitive, we report here on some of the performance results obtained in [44]. Specifically, we use the latest version of TruffleMATE to rerun two of the experiments presented in that work: i) a performance analysis of the VM when no metaobject is activated and ii) the performance of the immutable references scenario.

7.5.1 Experimental Setup

Most dynamic systems achieve their peak performance after a *warmup* phase in which the *hot code* is optimized. Since we are interested in the overheads a reflective VM incurs on its peak performance, we measure the run time of 50 iterations after the systems warmed up. We determined the end of the warmup phase manually by inspecting the run-time series plots for each configuration (benchmark + VM).

The reported values are run-time factors between TruffleMATE and a predefined baseline. To determine them, we take the mean run time for a specific iteration and use it as baseline. Our methodology is based on Kalibera et al. [68]. Note that they refer to this factor as the speed-up factor. Since we are measuring overheads instead of speed-ups we call it overhead factor (OF).

The benchmarking machine is a quad-core Intel Core i7-3770, 3.40 GHz with 16GB RAM and running the Linux kernel 4.2. For TruffleMATE we are using the precompiled binaries of Graal VM version 0.29

TABLE 2: Overall Results for the Read-only Benchmarks

Benchmark	OF	Confidence	Sd	Median	Min	Max
Proxies	4.26	±0.25	0.62	4.07	4.02	7.40
MOP	1.98	±0.15	0.46	1.82	1.80	4.19

7.5.2 Inherent Performance

To analyze the inherent overhead we use a set of VM benchmarks designed for comparing different language implementations [52]. Since these benchmarks were originally designed for being run by a standard VM, this experiment measures the overhead of supporting reflective capabilities on the VM even if they are not used. The suite includes micro and macro benchmarks measuring different language abstractions usually found on dynamic languages such as classical control flow operations, field reading/writing, and method dispatching.

To assess the inherent overhead we ran the benchmarks in TruffleSOM and TruffleMATE. To see how fast TruffleMATE is, we also ran the same benchmarks on two other industrial-strength VMs for dynamic languages: the CogVM version 6 for Pharo Smalltalk and Node.js version 8.9.1 using the Crankshaft compiler for JavaScript. We use as baseline the performance of the Java VM OpenJDK 1.8.0_91 with 64-Bit Server VM (25.91-b14).

Figure 5 presents boxplots of the overhead factors of each benchmark while Table 1 shows mean, median, confidence interval, and other statistical variables. Although TruffleMATE is a research prototype, it significantly outperforms Pharo. TruffleMATE also already performs similar to Node.js running on top of Google's optimized V8 VM. Finally, for this set of benchmarks, not using any behavior from the MOP, the overheads of using TruffleMATE instead of TruffleSOM are negligible.

The numbers indicate that TruffleMATE is slightly faster than TruffleSOM. This may be influenced by two main factors. First, the compilation and optimization of the benchmarks in TruffleMATE (including the intercession handling) results in different memory layouts. This could affect the cache-line hit rate. Moreover, languages implemented on top of Truffle/Graal are very sensitive to its inlining heuristics. The extra code of the intercession handling can result in different inline decisions potentially exposing additional optimization opportunities.

7.5.3 Immutable References

To assess the performance impact in applications using reflective capabilities, we took the read-only protection experiment from [44] which resembles the scenario presented in section 7.2.2. The benchmark essentially traverses a linked list, attempting to write some of its elements. The experiment compares two alternative approaches for read-only references: i) *handles* implemented using the reflective VM capabilities, and ii) *delegation proxies* [57] relying only on application-level reflection. The baseline uses a standard mutable reference.

It is worth noting that in both read-only cases, as the benchmark traverses each list element, it needs to wrap the reference to the next element of the list with either a handle or

a proxy. As a consequence, the read-only versions instantiate considerably more objects than the baseline.

Table 2 shows that the peak performance overhead of using the MOP is approximately 2x in comparison to the baseline. Notice that the MOP version is faster than the delegation proxies version which depends only on application-level reflective operations.

8 Discussion

TruffleMATE is an open-source reflective VM following the MATE architecture. As such, it demonstrates that it is feasible to implement a VM with advanced reflective capabilities. While more research is needed to assess advantages and drawbacks of reflection at the VM level, with our validation we argue that VMs supporting bidirectional communication between themselves and the applications are a suitable alternative for developing flexible software. Furthermore, the performance results indicates that it can be possible to remove most performance overheads of reflective VMs.

Regarding the adaptive scenarios, we compared TruffleMATE's approach with other language-level solutions and showed benefits in terms of our quality-based evaluation criteria. In all cases TruffleMATE dealt with the adaptation scenario by installing small metaobjects (about 20-45 LOC each) without changing the methods with application logic. It also usually ranked equal or better when considering most of our quality criteria. Finally, TruffleMATE handled all the cases while alternatives, such as AOP implementations or instrumentation frameworks, are only applicable to a subset.

A potential threat to the validity of our results is the set of quality attributes we selected. Furthermore, the coarse-grained values we defined could be considered fuzzy. To provide a more precise analysis, we would need to conduct user studies. This is challenging because it would require the availability of stable tools for each of the alternative approaches, which is not the case. Furthermore, the tools use various different conceptual abstractions. The lack of experts for all these abstractions and tools makes it even more challenging to perform a fair and consistent study. Another way to mitigate this threat would be to find proper quantitative metrics such as time or number of lines of code to apply the adaptations.

We are also aware that our empirical results may not generalize to every adaptation scenario. To mitigate this threat we have carefully selected examples of adaptation scenarios from existing literature. Moreover, we covered behavioral and structural adaptations at different abstraction levels. We compared our solution using TruffleMATE with other language-level approaches such as handles, AOP tools, and PBR. It is worth noticing that there exist few approaches able to address low-level adaptive scenarios at run time, and to the best of our knowledge, none is able to handle our whole set of experiments.

Finally, TruffleMate still lacks means for controlling the installation of metaobjects, leading to potential security issues. For instance, in the read-only protection case it would be desirable to allow only a privileged object to remove the metaobject of a handle. A capability-based layer on top of the VM-level reflective model could be used to provide such security guarantees.

9 Related Work

In this section we describe solutions from different domains that are related with the work presented in this paper.

9.1 Reflective Solutions

Pinocchio first class interpreter [33] is a practical implementation, in the context of an OO language, of Smith's tower of interpreters [27]. The interpreter is first-class and extensible from language level. In contrast to TruffleMATE, Pinocchio does not impose a fixed number of metalevels for dealing with metaregressions. It adapts to different levels on demand. On the other hand, Pinocchio is a reflective interpreter while TruffleMATE covers more VM-level entities. For instance, Pinocchio is not able to deal with the structural case studies of section 7.3, because it does not reify object layouts. Similar to Pinocchio, Asai [69] proposes a first-class interpreter but in the context of a functional language. It shares with Pinocchio the same fundamental differences with TruffleMATE.

CLOS [9] is an object-oriented layer for LISP that implements an advanced MOP, regarded as one of the most complete in terms of introspection and intercession reflective capabilities. CLOS reifies *Slots*, a language level representation of instance variables (fields). It also provides means to customize methods with generic functions, method combinators, and before/after methods. Since CLOS' main goal is enabling language customizations rather than being a reflective VM, it does not support extensive reflective capabilities for low-level functionalities such as the complete operational semantics of the language. Based on our understanding, using CLOS it may not be possible to handle the immutable references scenario of section 7.2.2 transparently because of its limitations for interceding the method lookup and activation on individual objects.

CodA proposes a metamodel that decomposes execution concepts into different roles. This decomposition is independent of the concrete programming language [47]. It is richer than the current TruffleMATE's MOP considering only the *message* metaobject because it reifies the receiver and sender roles separately and the synchronization of message activations. On the other hand, CodA reifications are similar to those of CLOS, more related to language-level concepts than to the VM perspective. For instance, CodA does not reify object layouts nor execution aspects such as method return values, or even the execution stack.

Flexible Object Layouts [22] reifies the internal structure of objects. Its main reification is the *Slot*, similar to slots in CLOS. Slots can be extended at run time by redefining four main operations: read, write, initialize and migrate. We followed a similar approach for implementing the `Layout` metaclass in TruffleMATE, just leaving out the migration operation which was not needed in our case studies.

9.2 Virtual Machines

Several self-hosted approaches for VM construction support some forms of VM-level reflection. Klein [6] for Self has similar goals to ours but its support for modifying VM-level entities at run time is not explained in the literature. The paper only mentions support for advanced mirror-based debugging tools to inspect and modify a remote VM.

Tachyon [7] translates the VM sources written in JavaScript to native code. Then, it uses special bridges for interacting with low-level entities of the VM. However, bridges are low-level mechanisms that only allow to call remote functions. Tachyon uses them to initialize a new VM during the bootstrap process. In contrast to a reflective VM, Tachyon was not designed with VM-level reflection as a goal and it does not provide advanced run-time adaptation capabilities of VM-level entities. Maxine [5] for Java, uses abstract and high-level representations of VM-level concepts and consistently exposes them throughout the development process. While debugging, the Maxine inspector provides a high degree of interactivity with the running VM at multiple abstraction levels. However, Maxine only allows to inspect but not to modify the VM at run time. Similarly, in the JikesRVM [4] components can be inspected but not modified at run time. Reflection on VM components is mainly used for the bootstrapping of the system. In contrast, a reflective VM focuses on providing interactivity during run time.

9.3 Dynamic Adaptations

To the best of our knowledge, Partial Behavioral Reflection (PBR) [42] is the most complete reflective solution for supporting unanticipated adaptations. PBR relies on bytecode instrumentation. Hence, it is restricted to adapting operational semantics. In addition, instrumentation techniques modify the application code and, from the VM perspective, the original code is not distinguishable from the instrumented code. In contrast, TruffleMATE fulfills the adaptations by using reified VM-level components and does not modify the application code. Concretely, TruffleMATE focuses on VM-level reflection while PBR depends on application-level reflection for (simulating) the low-level adaptations.

The Iguana/J environment [30] has capabilities similar to PBR. However, it provides these capabilities with a MOP similar to ours in terms of behavioral adaptive capabilities. Similar to CLOS, Iguana/J provides intercession handlers for method interceptions, reading, and writing of fields. TruffleMATE allows to intercept a broader set of operations such as the complete operational semantics of the system and provides structural VM-level reflective capabilities.

Aspect-oriented programming [28] (AOP) is used to introduce changes into software systems with focus on crosscutting concerns rather than on reflecting on the system. Most AOP implementations provide a domain specific language (DSL) to specify a set of points in the program (*join points*) at which a feature orthogonal to the application logic such as logging, caching, and persistence must be executed. An *aspect weaver* embeds the so-called *cross-cutting concerns* (*advices*) at the corresponding joinpoints either statically or at run time into the program.

Context-oriented programming (COP) is a paradigm specially designed for applications with behavioral variations depending on contextual information [29]. In COP terms, context means any computationally accessible information. Consequently, COP provides abstractions for expressing contextual conditions. In the absence of such constructs, application logic would become tangled with the needed adaptations. COP could be considered as a specialized form of AOP introducing *context-aware aspects*.

To handle scenarios such as those included in our running examples, AOP and COP frequently required indirect mechanisms of adaptation. The reason is mainly that they were not conceived as general solutions for unanticipated software adaptation. As a consequence, they do not generally provide means to directly customize low-level features. In contrast, they promote mechanisms (*e.g.*, pointcut languages, layers) for supporting their original goals, and thus, biased the user to think in terms of intercepting execution points and redirecting their execution flow. While AOP can approach the profiling scenario directly, it can neither express direct adaptations for the read-only protection scenario nor the two structural adaptation cases.

To support run-time adaptation and fine-grained scoping, AOP implementations use dynamic weaving. To do so efficiently they utilize dedicated VM support [61], [62], [63]. Concretely, the VM must provide some kind of intercession handling. For instance Steamloom [61] does so by providing a dedicated compiler for instrumenting bytecodes. This enables Steamloom to install intercession handlers, at run time, only in the locations that actually need the adaption. From this perspective, the ubiquitous intercession handling of a reflective VM generalizes AOP's intercession handling. Nevertheless, the preliminary evaluation suggests that our more general approach could also result in negligible overheads (see Section 7.5).

Cazzola et al. [70] recently proposed to move the evolution from the application level to the programming language level to support direct dynamic adaptations. They use Neverlang [71], a micro-language framework for modular language development. Their goals are similar to ours: separation of concerns, maintainability, and reuse of adaptations. However, in contrast to a reflective VM their adaptation abstractions are at the programming language and not the VM level. On the one hand, the approach enables adaptations that a reflective VM does not provide. This includes the direct customization of arithmetic operations or control flow operators, such as loop, even for single instances. On the other hand, it does not enable developers to directly customize object layouts, execution stack, memory organization, or other VM-level aspect. Consequently, the approaches are complementary. Moreover, since Neverlang is based on micro operations described using grammars, grammars are also the means to provide the adaptations. In contrast, a reflective VM is based on MOPs and stays within the application's programming language syntax and semantics.

10 Conclusions

In this paper we introduce the concept of fully reflective execution environments and discuss the main properties they should fulfill.

In order to validate the feasibility and to understand the overall potential of a fully reflective execution environment, we designed and implemented TruffleMATE: a fully functional Smalltalk VM featuring a dynamic compiler and advanced reflective capabilities at the VM level. TruffleMATE enables the introspection and intercession of behavioral and structural VM concepts at run time. The degree of reflection reached by TruffleMATE is a good indicator for the feasibility of VMs with extensive reflective capabilities.

Moreover, our empirical evaluation assessed the potential of TruffleMATE for handling a series of *unanticipated fine-grained adaptation* scenarios on the fly. We compared solutions based on TruffleMATE against alternative solutions considering a wide range of qualitative aspects. The results showed that TruffleMATE can handle a series of heterogeneous scenarios covering structural and behavioral low-level adaptive requirements by using run-time reflection. In all the cases, the solution in TruffleMATE consisted of few lines of code (between 20-45) and presented benefits over the existing alternatives. Furthermore, we are not aware of any other platform capable of dealing with all scenarios.

In addition to its potential for dealing with concrete software problems, we think that the provided evidence suggests that reflective VMs open new avenues for software development that needs to deal with evolution at run time. Moreover, the first performance indications are encouraging, but more work is needed to further improve the performance when the full power of reflective VMs is used.

Future Work. TruffleMATE does not reify the `Memory` component and, thus, we can neither directly adapt the memory management nor the garbage collector. We plan to explore the impact of providing reflective capabilities to this component. We also plan to provide more extensive reflective capabilities to the `Executor`, *e.g.*, to support a trace-based execution schema that could provide history-aware semantics to the applications.

In general, fundamental limits of VM-level reflection, both in terms of feasibility and performance impacts, still need to be explored further. Questions that need to be addressed are, for instance, what the minimal core of a reflective VM is that cannot be reified, or more practically, what the consequences of reifying specific VM components and operations are. With incorporating more reflective capabilities, we expect to encounter new challenges like stronger causal connections or different performance issues. Some of them may require modeling reflection in new or different ways. For instance, the ideas implemented in high-level low-level programming frameworks such as *Benzo* [72] and *org.vmmagic* [73] may be suitable for supporting reflective capabilities for the memory component.

To assess the impact of reflective VMs in more detail, a set of quantitative metrics is needed to precisely distinguish the reflectivity of different solutions incorporating VM-level reflection. Classical models of reflection like Smith's et al. [27] and the denotational semantics presented by Wand and Friedman [74] do not distinguish reflective capabilities at a fine-grained level. Thus, a novel formalization model to fulfill our requirements is needed. User studies to investigate how developers manage to handle the case studies with the different approaches could provide further insights into the utility of reflective VMs and whether they live up to the high expectations we have in them.

Acknowledgments

We would like to thank the anonymous reviewers, as well as Laurence Tratt, for their constructive feedback which contributed to significantly improve this paper. This work was partially supported by the projects, ANPCYT PICT 2013-2341, ANPCYT PICT 2014-1656, ANPCYT PICT 2015-1718, UBA-CYT 384, CONICET PIP 2014/16 No11220130100688CO, CONICET PIP 2015/17 No11220150100931CO. Stefan Marr was partially funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31 while working at the Johannes Kepler University Linz, Austria.

References

[1] T. Mens and S. Demeyer, *Software Evolution*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[2] G. Polito, S. Ducasse, L. Fabresse, N. Bouraqadi, and M. Mattone, "Virtualization support for dynamic core library update," in *Onward! 2015*, 2015. [Online]. Available: http://rmod.inria.fr/archives/papers/Poli15b-Onward-CoreLibrariesHotUpdate.pdf

[3] L. Baresi and C. Ghezzi, "The disappearing boundary between development-time and run-time," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. ACM, 2010, pp. 17–22. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882367

[4] 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, and V. Sarkar, "The Jikes Research Virtual Machine Project: Building an Open-source Research Community," *IBM Syst. J.*, vol. 44, no. 2, pp. 399–417, Jan. 2005. [Online]. Available: http://dx.doi.org/10.1147/sj.442.0399

[5] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, Java," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 30:1–30:24, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400689

[6] D. Ungar, A. Spitz, and A. Ausch, "Constructing a metacircular virtual machine in an exploratory programming environment," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. ACM, 2005, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1094855.1094865

[7] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour, "Bootstrapping a Self-hosted Research Virtual Machine for JavaScript: An Experience Report," in *Proceedings of the 7th Symposium on Dynamic Languages*, ser. DLS '11. ACM, 2011, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2047849.2047858

[8] G. Chari, D. Garbervetsky, S. Marr, and S. Ducasse, "Towards fully reflective environments," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 240–253. [Online]. Available: http://doi.acm.org/10.1145/2814228.2814241

[9] G. Kiczales and J. D. Rivieres, *The Art of the Metaobject Protocol*. MIT Press, 1991.

[10] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, "Self-optimizing ast interpreters," in *DLS*. ACM, 2012, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/2384577.2384587

[11] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kie, un, and M. D. Ernst, "Object and Reference Immutability Using Java Generics," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. ACM, 2007, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287637

[12] M. S. Tschantz and M. D. Ernst, "Javari: Adding Reference Immutability to Java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. ACM, 2005, pp. 211–230. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094828

[13] D. Holten, B. Cornelissen, and J. J. van Wijk, "Trace visualization using hierarchical edge bundles and massive sequence views," *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, vol. 0, pp. 47–54, 2007.

[14] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP '08. Berlin, Heidelberg:

Springer-Verlag, 2008, pp. 542–565. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70592-5_23

[15] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic Metrics for Java," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 149–168. [Online]. Available: http://doi.acm.org/10.1145/949305.949320

[16] D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz, "Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks," *IEEE Trans. Softw. Eng.*, vol. 38, no. 3, pp. 579–591, May 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.42

[17] J. Whaley, "A Portable Sampling-based Profiler for Java Virtual Machines," in *Proceedings of the ACM 2000 Conference on Java Grande*, ser. JAVA '00. New York, NY, USA: ACM, 2000, pp. 78–87. [Online]. Available: http://doi.acm.org/10.1145/337449.337483

[18] C. Chambers, D. Ungar, and E. Lee, "An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes," in *OOPSLA*. ACM, October 1989, pp. 49–70.

[19] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck, "An object storage model for the truffle language implementation framework," in *PPPJ*. ACM, 2014, pp. 133–144.

[20] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen, "Read-only execution for dynamic languages," in *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, ser. TOOLS'10. Springer-Verlag, 2010, pp. 117–136. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894386.1894393

[21] T. Mattis, J. Henning, P. Rein, R. Hirschfeld, and M. Appeltauer, "Columnar objects: Improving the performance of analytical applications," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 197–210. [Online]. Available: http://doi.acm.org/10.1145/2814228.2814230

[22] T. Verwaest, C. Bruni, M. Lungu, and O. Nierstrasz, "Flexible object layouts: Enabling lightweight language extensions by intercepting slot access," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048138

[23] G. Chari, D. Garbervetsky, and S. Marr, "Fully-reflective VMs for Ruling Software Adaptation," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 229–231. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.144

[24] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/258915.258924

[25] H. Plattner, "A common database approach for oltp and olap using an in-memory column database," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559846

[26] ——, *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Springer Publishing Company, Incorporated, 2013.

[27] B. C. Smith, "Reflection and Semantics in LISP," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '84. ACM, 1984, pp. 23–35. [Online]. Available: http://doi.acm.org/10.1145/800017.800513

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242. [Online]. Available: https://doi.org/10.1007/BFb0053381

[29] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented programming," *Journal of Object Technology*, vol. 7, no. 3, 2008.

[30] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behaviour," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, ser. ECOOP '02. Springer-Verlag, 2002, pp. 205–230. [Online]. Available: http://dl.acm.org/citation.cfm?id=646159.680029

[31] D. Röthlisberger, M. Denker, and E. Tanter, "Unanticipated partial behavioral reflection: Adapting applications at runtime," *Comput. Lang. Syst. Struct.*, vol. 34, no. 2-3, pp. 46–65, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2007.05.001

[32] M. Haupt, C. Gibbs, B. Adams, S. Timbermont, Y. Coady, and R. Hirschfeld, "Disentangling virtual machine architecture," *Software, IET*, June 2009.

[33] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, and O. Niestrasz, "Pinocchio: Bringing reflection to life with first-class interpreters," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. ACM, 2010, pp. 774–789. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869522

[34] P. Maes, "Concepts and experiments in computational reflection," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '87. ACM, 1987, pp. 147–155. [Online]. Available: http://doi.acm.org/10.1145/38765.38821

[35] E. Tanter, "Reflection and open implementations," DCC, University of Chile, Tech. Rep., 2009. [Online]. Available: http://www.dcc.uchile.cl/TR/2009/TR_DCC-20091123-013.pdf

[36] J.-P. Briot and P. Cointe, "Programming with Explicit Metaclasses in Smalltalk-80," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '89. New York, NY, USA: ACM, 1989, pp. 419–431. [Online]. Available: http://doi.acm.org/10.1145/74877.74921

[37] G. Attardi, C. Bonini, M. R. Boscotrecase, T. Flagella, and M. Gaspari, "Metalevel Programming in CLOS." in *ECOOP*, vol. 89, 1989, pp. 243–256.

[38] P. Cointe, "Metaclasses Are First Class: The ObjVlisp Model," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '87. New York, NY, USA: ACM, 1987, pp. 156–162. [Online]. Available: http://doi.acm.org/10.1145/38765.38822

[39] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 485–495. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337280

[40] J. Ressia, L. Renggli, T. Gîrba, and O. Nierstrasz, "O.: Run-time evolution through explicit meta-objects," in *In: Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS*, 2010, pp. 37–48.

[41] G. Bracha and D. Ungar, "Mirrors: Design principles for meta-level facilities of object-oriented programming languages," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '04. ACM, 2004, pp. 331–344. [Online]. Available: http://doi.acm.org/10.1145/1028976.1029004

[42] E. Tanter, J. Noyé, D. Caromel, and P. Cointe, "Partial behavioral reflection: Spatial and temporal selection of reification," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03. ACM, 2003, pp. 27–46. [Online]. Available: http://doi.acm.org/10.1145/949305.949309

[43] S. Marr, C. Seaton, and S. Ducasse, "Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015. ACM, 2015, pp. 545–554. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737963

[44] G. Chari, D. Garbervetsky, and S. Marr, "Building efficient and highly run-time adaptable virtual machines," in *Proceedings of the 12th Symposium on Dynamic Languages*, ser. DLS 2016. New York, NY, USA: ACM, 2016, pp. 60–71. [Online]. Available: http://doi.acm.org/10.1145/2989225.2989234

[45] B. Meyer, *Object-oriented Software Construction*. Prentice-Hall, Inc., 1997.

[46] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[47] J. McAffer, "Meta-level Programming with CodA," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ser. ECOOP '95. London, UK, UK: Springer-Verlag, 1995, pp. 190–214. [Online]. Available: http://dl.acm.org/citation.cfm?id=646153.679534

[48] S. Chiba, G. Kiczales, and J. Lamping, "Avoiding confusion in metacircularity: The Meta-Helix," in *Proceedings of the Second JSSST*

International Symposium on Object Technologies for Advanced Software, ser. ISOTAS '96. Springer-Verlag, 1996, pp. 157–172. [Online]. Available: http://portal.acm.org/citation.cfm?id=646898.756984
[49] M. Denker, M. Suen, and S. Ducasse, *The Meta in Meta-object Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 218–237. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69824-1_13
[50] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to Rule Them All," in *Onward!* ACM, 2013, pp. 187–204.
[51] S. Marr and S. Ducasse, "Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters," in *OOPSLA*. ACM, 2015, pp. 821–839.
[52] S. Marr, B. Daloze, and H. Mössenböck, "Cross-Language Compiler Benchmarking—Are We Fast Yet?" in *Proceedings of the 12th Symposium on Dynamic Languages*, ser. DLS'16. ACM, 2016.
[53] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 662–676. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062381
[54] M. Haupt, R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn, "The SOM Family: Virtual Machines for Teaching and Research," in *ITiCSE*. ACM, 2010, pp. 18–22. [Online]. Available: http://www.hpi.uni-potsdam.de/hirschfeld/publications/media/HauptHirschfeldPapeGabrysiakMarrBergmannHeiseKleineKrahn_2010_TheSomFamily_AcmDL.pdf
[55] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '97. ACM, 1997, pp. 318–326. [Online]. Available: http://doi.acm.org/10.1145/263698.263754
[56] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009. [Online]. Available: http://pharobyexample.org
[57] E. Wernli, O. Nierstrasz, C. Teruel, and S. Ducasse, "Delegation proxies: The power of propagation," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14. ACM, 2014, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2577080.2577081
[58] M. Dmitriev, "Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation," Mountain View, CA, USA, Tech. Rep., 2003.
[59] A. Bergel, F. Bañados, R. Robbes, and D. Röthlisberger, "Spy: A flexible code profiling framework," *Comput. Lang. Syst. Struct.*, vol. 38, no. 1, pp. 16–28, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2011.10.002
[60] A. Villazón, W. Binder, P. Moret, and D. Ansaloni, "Comprehensive Aspect Weaving for Java," *Sci. Comput. Program.*, vol. 76, no. 11, pp. 1015–1036, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2010.04.007
[61] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann, "Virtual Machine Support for Dynamic Join Points," in *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, ser. AOSD '04. New York, NY, USA: ACM, 2004, pp. 83–92. [Online]. Available: http://doi.acm.org/10.1145/976270.976282
[62] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, *An Overview of CaesarJ*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 135–173. [Online]. Available: https://doi.org/10.1007/11687061_5
[63] R. Dyer and H. Rajan, "Nu: A Dynamic Aspect-oriented Intermediate Language Model and Virtual Machine for Flexible Runtime Adaptation," in *Proceedings of the 7th International Conference on Aspect-oriented Software Development*, ser. AOSD '08. New York, NY, USA: ACM, 2008, pp. 191–202. [Online]. Available: http://doi.acm.org/10.1145/1353482.1353505
[64] B. Dufour, L. Hendren, and C. Verbrugge, "*J: A Tool for Dynamic Analysis of Java Programs," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 306–307. [Online]. Available: http://doi.acm.org/10.1145/949344.949425
[65] M. Arnold and D. Grove, "Collecting and exploiting high-accuracy call graph profiles in virtual machines," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 51–62. [Online]. Available: http://dx.doi.org/10.1109/CGO.2005.9
[66] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, "Accurate, efficient, and adaptive calling context profiling," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 263–271. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134012
[67] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1108970.1108971
[68] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. New York, NY, USA: ACM, 2013, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/2464157.2464160
[69] K. Asai, "Reflection in direct style," in *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, ser. GPCE '11. ACM, 2011, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/2047862.2047882
[70] W. Cazzola, R. Chitchyan, A. Rashid, and A. Shaqiri, "μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating," *Computer Languages, Systems & Structures*, 2017.
[71] E. Vacchi and W. Cazzola, "Neverlang: A framework for feature-oriented language development," *Computer Languages, Systems & Structures*, vol. 43, pp. 1–40, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2015.02.001
[72] C. Bruni, S. Ducasse, I. Stasenko, and G. Chari, "Benzo: Reflective Glue for Low-level Programming," in *International Workshop on Smalltalk Technologies*, Aug. 2014. [Online]. Available: https://hal.inria.fr/hal-01060551
[73] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev, "Demystifying magic: High-level low-level programming," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09. ACM, 2009, pp. 81–90. [Online]. Available: http://doi.acm.org/10.1145/1508293.1508305
[74] M. Wand and D. P. Friedman, "The mystery of the tower revealed: A non-reflective description of the reflective tower," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ser. LFP '86. ACM, 1986, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/319838.319871

Guido Chari is a doctoral candidate in computer science at the Department of Computing, FCEyN, Universidad de Buenos Aires, where he is also a teaching assistant. His research interests include programming languages design and implementation. In particular, providing tools and methods to facilitate program development, evolution, and maintenance.

Diego Garbervetsky holds a Professorship at the Department of Computing, FCEyN, Universidad de Buenos Aires and he is a CONICET researcher working in the area of Software Engineering. His research interests are program analysis and optimization, focused on the inference of quantitative and qualitative properties of programs. He has participated in several European projects and also been awarded by Microsoft Research and IBM Eclipse Innovation program.

Stefan Marr is a lecturer at the University of Kent. He investigates how to safely combine concurrency models and how to implement complex language semantics efficiently. In his PhD thesis, he proposed an ownership-based metaobject protocol as a unifying substrate for concurrency support in multi-language VMs.

Stéphane Ducasse is a research director at INRIA Lille leading the RMoD Team. During 10 years, he co-directed with Oscar Nierstrasz the Software Composition Group. He is also president of ESUG and co-founded Synectique, a company that offers specific tools for Software analysis.