Towards Fully Reflective Environments

Guido Chari, Diego Garbervetsky
Departamento de Computación, FCEyN, UBA
and CONICET, Argentina
{gchari,diegog}@dc.uba.ar

Stefan Marr, Stéphane Ducasse
RMoD, INRIA Lille - Nord Europe, France
mail@stefan-marr.de, stephane.ducasse@inria.fr

Abstract

Modern development environments promote live programming (LP) mechanisms because it enhances the development experience by providing instantaneous feedback and interaction with live objects. LP is typically supported with advanced reflective techniques within dynamic languages. These languages run on top of Virtual Machines (VMs) that are built in a static manner so that most of their components are bound at compile time. As a consequence, VM developers are forced to work using the traditional edit-compile-run cycle, even when they are designing LP-supporting environments. In this paper we explore the idea of bringing LP techniques to the VM domain for improving their observability, evolution and adaptability at run-time. We define the notion of fully reflective execution environments (EEs), systems that provide reflection not only at the application level but also at the level of the VM. We characterize such systems, propose a design, and present Mate v1, a prototypical implementation. Based on our prototype, we analyze the feasibility and applicability of incorporating reflective capabilities into different parts of EEs. Furthermore, the evaluation demonstrates the opportunities such reflective capabilities provide for unanticipated dynamic adaptation scenarios, benefiting thus, a wider range of users.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Interactive Environments; D.3.4 [*Processors*]: Run-time Environments

General Terms Design, Languages, Experimentation

Keywords Reflection, Live Programming, Virtual Machines, Metaobject-Protocols, Dynamic Adaptation

1. Introduction

With the recently renewed interest in *live programming* (LP), it is becoming more popular to build software in environments that assist developers with immediate and continuous feedback. Such environments blur the boundary between development time and run-time, which is not only beneficial for prototyping tasks, but also for developing complex software that needs exploration and continuous evolution [4]. In dynamic languages such as JavaScript, Python, or Smalltalk, LP is typically enabled by language-level mechanisms for observability and interactivity in the form of *reflective* APIs.

Dynamic languages that facilitate LP usually run on top of virtual machines (VMs). VMs are, in general, highly complex software artifacts that realize heterogeneous functionalities such as the language's semantics, dynamic compilation, adaptive optimizations, memory management, and security enforcement. However, these artifacts are typically developed using tools with a strict edit-compile-run cycle that do not provide the aforementioned dynamic features.

For example, industrial-strength VMs for mainstream languages like Java and JavaScript are written in low-level static languages, such as C and C++, to comply with performance requirements. After compilation, such VMs are optimized binaries that make it hard to observe, explore, and adapt their behavior at run-time. Thus, while application developers benefit from LP capabilities, VM developers still live in the old-fashioned static world, where build times can be in the order of minutes rather than milliseconds, significantly slowing down the development process.

Multiple research projects explored the use of high-level languages for VM construction [1, 11, 30, 36]. This approach is appealing because developers can leverage modern programming techniques and principles to better deal with the complexity of VM's development. Some of these approaches are metacircular, *e.g.*, implementing a Java Virtual Machine in Java, and thus benefit from the language's features. In addition to the use of high-level languages, aspects such as modularity, observability and extensibility have been in the focus of the VM

research community as well [13, 35]. However, even the metacircular approaches produce VMs that do not enable significant observability and interactivity with the VM at run-time.

We propose the idea of fully reflective EEs: VMs exposing their whole structure and behavior to the language at run-time. Fully reflective EEs allow developers to observe and adapt the VM *on-the-fly* enabling from simple adaptations up to fine-grained tuning of applications. This benefits VM developers by bringing them the possibility to program low-level components with instantaneous feedback, something rarely available nowadays. At the same time, it also provides end-users a high-level interface for customizing the EE. For instance, application developers can rely on run-time services for adaptability instead of having to develop application-specific solutions.

As an example for the potential of such VMs, consider an 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. To avoid further problems, it is desirable to ensure that this data is not modified by the analysis, *i.e.*, that it is side-effects free. A programming language approach for enforcing this property would turn the critical data of the application immutable before running the analysis. In this paper we argue that solutions based on fully reflective EEs increase the possibilities to approach such scenarios at the language level while also simplifying the solutions.

The goal of this paper is start exploring the space of fully reflective EEs. We begin by defining their main characteristics and we design a reference architecture to follow. Then, we implement a first prototype, called Mate v1, that exposes a significant part of its structure and behavior using a language-level uniform abstraction: a metaobject protocol (MOP) [16]. To the best of our knowledge, our approach is the first that uses reflection at EE level in an integral way. Furthermore, Mate v1 provides extensive reflective capabilities in most of its main components. To assess the feasibility, applicability, and usefulness of our approach we analyze how the EE handles unanticipated fine-grained adaptive scenarios concerning low-level aspects. We conclude that, using its reflective capabilities, Mate v1 properly deals with the required modifications *on-the-fly*.

The contributions of this paper are:

- The proposal of bringing live programming techniques to the domain of EEs together with a methodology to study and characterize fully reflective EEs in order to explore their advantages and limitations.
- A reference architecture for fully reflective EEs featuring a MOP for handling EE-level reflection at the application level.
- Mate[1]: a self-hosted prototypical, but functional, reflective EE supporting the Smalltalk programming language.
- A validation through case studies in the context of dynamic adaptation, that demonstrates the feasibility and usefulness of our approach by using the incorporated LP capabilities at the EE level.

2. Background

This section introduces basic notions on which we rely throughout the paper.

2.1 Reflection

Reflection [27] 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*. A programming language is said to have a reflective architecture if it provides tools for reflective computations explicitly [17]. For instance, a reflective architecture for OO languages can rely on metaobjects that reify language concepts such as objects and methods. Metaobjects and their corresponding baselevel objects must be *causally connected*: changes in any of them must lead to a corresponding effect upon the other [17].

Metaobject protocols (MOP) [16] are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation within the same language. To improve aspects of distribution, deployment, and general purpose metaprogramming for MOPs, Bracha and Ungar [7] proposed the following *Mirror* design principles: i) *Encapsulation:* they do not expose implementation details. ii) *Stratification:* they enforce a separation between the application behavior and the reflective code. iii) *Ontological correspondence:* their meta-level reifications must map one-to-one to concepts of the base-level domain. Since these principles also correspond to common programming practices, we base our design on them.

2.2 Reflection Challenges

Reflective implementations must deal with two main concerns that are in tension: *completeness* and *performance*. Completeness is the degree in which the concepts of a domain (i.e., components and their responsibilities) are reified. We also call this degree the reflectivity of the system. Within completeness we can distinguish two main dimensions: domain-breadth and domain-depth. The former measures how many entities and their corresponding functionalities are included in the reification. The latter refers to the number of meta-levels that can be used from the domain. *Full reflection* refers to the

[1] https://ci.inria.fr/rmod/view/Mate/job/MateArg/

coverage of both the domain-bread and domain-depth aspects of reflection. Completeness and performance are in tension because incorporating more reflection into a system (i.e., making it more complete) increases the flexibility at the cost of affecting its performance.

Within completeness, the domain-depth dimension is strongly related with the concept of *metaregression*. A well-known example from literature that exposes this situation is the tower of interpreters [27]: when an interpreter is reified it requires another (not yet reified) entity to interpret itself. This scales to an infinite tower where each level is in charge of giving semantics to the subsequent upper level. The lowest level, *i.e.*, the base level, is the application. In practice, there are different ways to avoid this infinite chain of meta-activations. For instance, one approach is fixing the number of metalevels so that the top level can not be further reified. Note that this alternative limits domain-depth completeness. All alternatives face with similar trade-offs. As a consequence, it is infeasible to reify everything considering both domain-breadth and domain-depth.

2.3 Live Programming

Live programming is mainly concerned with providing instantaneous feedback to developers. From a conceptual point of view, a live interaction helps to better understand and manage complex problems. LP is also suitable for exploratory stages since it speeds up development by reducing offline compilation steps [10, 20]. In particular, Burckhardt et. al [10] see the classical edit-compile-run cycle as one reason for the gap between the program text and the perception of its effects.

The elimination of this cycle is already supported by high-level OO programming languages, such as Smalltalk, Ruby, Javascript, or Python, via reflective APIs. For instance Smalltalk, with its reflective object model, embraces a *fix-and-continue* way of debugging, where code and state can be modified while the program is being debugged. Similarly, in JavaScript objects behave like hash maps so that fields can be dynamically added or removed with instantaneous effects.

2.4 Execution Environments

We define an *Execution Environment* to be any layer of software within a system that is responsible for executing programs written in a specific programming language. Particularly, in this work we are interested in EEs for object-oriented (OO) languages. For instance, an EE is responsible for executing language expressions, define the representation of objects in memory, and garbage collect objects. It is worth noting that many EEs are also known as Virtual Machines (VMs) or Managed Runtimes. We use the terms interchangeably throughout this paper.

A common characteristic of EEs is that they consist of several intertwined components coping with complex and low-level responsibilities [14]. In addition, their performance affects the overall performance of the programs they execute. We already mentioned in section 1 that as a consequence, they are usually rather static artifacts, difficult to observe and adapt at run-time. For instance, a VM developer that wants to experiment with another algorithm for the *method lookup* mechanism, must recompile and deploy the whole VM. A system supporting LP at EE-level would allow her to change the method lookup in a programmatic manner and instantaneously analyze its effects.

2.5 Application-level vs. EE-level Reflection

Commonly, reflective computations are distinguished based on whether they use introspection or intercession, as well as whether they affect behavioral or structural elements. However, this does not distinguish reflective operations at different abstraction levels. For example the operations to add fields to an object and to modify its memory position are both characterized as structural intercession, but they deal with different levels of abstraction. The first operation is at the object (application) level while the latter is at the EE level. Moreover, it is common in reflective languages, such as Smalltalk and Javascript, to support the addition of fields at run-time while most languages do not support the modification of the memory position of an object. For the work discussed in this paper, it is important to distinguish this kind of operations to precisely characterize reflective environments. As a consequence, 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.
- *EE-level reflection* refers to metaprograms that regard operational semantics, execution stack, layout, method lookup, or memory management.

3. Exploring Fully Reflective EEs

In this section we describe the characteristics that EEs must fulfill for being *fully reflective*. We also present the research questions we identified after analyzing them. Finally, we describe our methodological approach to answer these questions.

3.1 Main Characteristics

The following maxims define, from our perspective, the main characteristics that *fully reflective* EEs must have.

Maxim 1. Universal Reflective Capabilities:

EEs must enable intercession and introspection of all entities at both, the application and the EE level.

As we already pointed out, EEs for dynamic languages are generally written in low-level languages. As a consequence, EE's developers lack the LP capabilities that the application level promotes. Moreover, EEs usually impose a rigid boundary between them and the application, beneficial in terms of security and portability but severely restricting the interaction between an application and its EE at run-time. We propose to push LP techniques to the domain of EEs. More precisely, we advocate for EEs with reflective capabilities that cover the cartesian product of the dimensions introduced in section 2.5:

$$\{Intercession, Introspection\} \times \{Structure, Behavior\} \times \{Application, Environment\}$$

Maxim 2. Uniform Reflective Abstractions:

EEs must provide the same language tools for interacting with both, the application and the EE levels.

Uniform tools for logically related concepts help to improve the understandability and evolvability of the programming environment [21]. Hence, improving the reflective capabilities with EE-level reflection must use the same application-level reflective techniques to avoid increasing the complexity. Developers that work in different domains should be able to focus on enhancing their knowledge on a single tool for dealing with reflective computations at different levels. For instance, if the language offers application-level reflection via a MOP, EE-level reflection must also be supported by a MOP.

Maxim 3. Separation of Application and EE:

Observability and adaptability should not be a concern of an application's design. Instead, the EE 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 making objects immutable 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 EE domains. Moreover, EE-level reflective capabilities must not impose any cost when not used.

3.2 Analysis of Fully Reflective EE

Our research goal is to understand the consequences of incorporating reflective capabilities in all EE components. This includes analyzing the classical issues of completeness and performance, discussed in section 2.2, but also the effects derived from the particular characteristics of the EE domain. To approach this goal we intend to answer a series of questions regarding feasibility, performance and applicability.

Starting with feasibility, we need to understand the potentials and limitations of modeling full reflection at the EE domain. In section 2.2 we already discussed that full reflection is not feasible because of the metaregression problem [18] for the domain-depth dimension of completeness. But, we are not aware of previous works exploring the reflective capabilities of EEs in a domain-breadth fashion. We think a finer-grained analysis in the breadth is particularly relevant for the EE domain. The reason is that EEs include complex elements that are highly coupled and, thus, reflective implementations must ensure causal relationships with more encompassing implications. For instance, changes to a GC property potentially need to be taken into account by a JIT compiler to generate different code. Taking all this into account we propose the following research questions:

- *What are the precise fundamental limits of fully reflective EEs?*
- *What is the minimal core of an EE that cannot be implemented in a fully reflective way?*
- *What are the techniques for dealing with the fundamental limitations?*
- *Do reflective capabilities in one component interfere with the capabilities of others, and if so how?*

In the context of applicability, we think it is also important to analyze pragmatical issues such as the relevance of fully reflective EEs in the context of real applications. For instance, understand how design decisions impact on the capabilities for properly handling different problems at run-time. Furthermore, practical EEs must meet certain performance demands. Although reflection imposes significant performance overheads [8, 18], Marr et al. [19] recently showed that it is possible to remove this overhead of language-level reflection in the context of just-in-time compilation. In summary, we are interested in studying also usability and performance aspects of fully reflective EEs such as:

- *Are fully reflective EEs suitable for tackling realistic problems?*
- *What are the reflective operations that are better suited for each kind of scenario?*
- *What are the main performance overheads of fully reflective EEs? How can they be mitigated?*

Finally, there exists a potential abstraction mismatch between the high-level nature of the language and the

low-level nature of the EE. For instance, it is not clear how to express computations for handling memory issues using a high-level language that does not provide constructs for manually managing memory (e.g., it uses a garbage collector). This is something a fully reflective EE must address:

- *Is there a proper way to deal with the abstraction mismatch between the language expressiveness and the environment low-level necessities?*

3.3 Approach

In order to approach the research questions in a systematic manner we use the following methodology:

1. Define a reference architecture.
2. Inspired by a representative problem, design and implement an EE prototype that increments the reflective capabilities of the previous iteration.
3. Analyze the reflective capabilities and the ability to address the case studies of the prototype.
4. Incorporate the feedback of the empirical data into the analysis of the research questions.
5. If there remain unanswered aspects go back to 2.

In the remainder of this paper we describe our reference architecture in section 4 and evaluate it in two phases. In section 5 we select a representative problem and then discuss the design, implementation and analysis of the corresponding EE prototype. We use this implementation for gathering insights about potential feasibility issues and studying the characteristics of different reflective capabilities. Section 6 describes the second part, an evaluation of the prototype in the context of a series of case studies. This allows us to gather information about applicability aspects. Finally, with all the information obtained, in section 7 we answer (at least partially) the research questions.

4. Reference Architecture

Recall from section 3.1 that we aim to design a *universal reflective EE* that exposes a *uniform reflective abstraction* which promotes *separation of the application and EE domains*. Guided by these maxims we designed an architecture following these three guidelines:

1. The EE supports a language already providing extensive reflective capabilities at the application level.
2. Every EE-level entity features reflective capabilities.
3. EE-level reflection is realized using an independent MOP that follows the Mirror's principles.

Figure 1 presents the resulting architecture that forms the basis for the implementation of the successive prototypes. The architecture is divided into two layers:

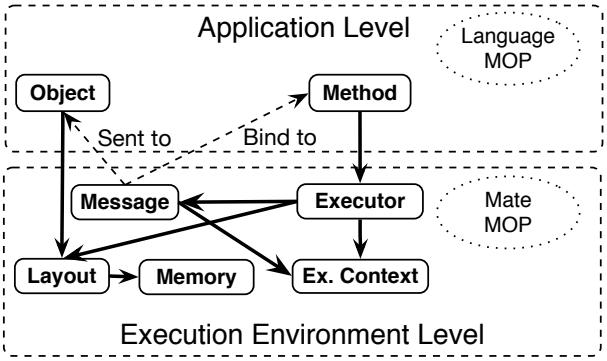


Figure 1: Mate High-Level Architecture.

the application and the EE levels. The arrows represent interaction points between components. The application level includes only the fundamental blocks of the OO programing paradigm: *objects* and *methods*. The idea is to minimize the restrictions imposed so that more existing languages fit in the architecture. The only requirement for the OO language is that it must include a MOP in charge of the application-level reflection, which to a certain degree most OO solutions do to realize their reflective architectures.

The bottom layer comprises a set of essential EE-level entities needed for executing expressions, realizing objects, and managing memory in a pure OO language. Following universal reflection (maxim 1) they must all be first-class citizens in any implementation of the architecture. Moreover, following uniformity (maxim 2) we decided to structure EE-level reflection as a MOP complementary to the application-level MOP. The bottom rounded dashed box in figure 1 shows this graphically. We based our decision on the fact that several authors suggested that MOPs are an elegant solution for handling non-functional aspects and we have the hypothesis that MOPs also fit well with our requirements. In fact, there are already approaches such as Iguana/J [23], Object-Centric Debugging [24] and Slots [32], that adopt MOPs as a way to deal with low-level concerns.

Finally, MOPs adhering to the Mirror's principles isolate the reflective capabilities into separate intermediary objects that directly correspond to language structures. The requirement of adhering to these principles at the EE level provides the required separation of domains of maxim 3. The explicit separation we impose between EE-level and application-level MOPs already comply with the Mirror's principle of *stratification*. Bracha and Ungar claim that adhering to this principle helps to avoid overheads when EE-level reflection is not needed by making it easy to eliminate it [7]. In addition, the decision of representing each EE-level entity by a metaclass honors the Mirror's principle of *ontological correspondence*.

We now describe briefly the main responsibilities of the EE-level entities included in our reference architecture:

Executor is responsible for interpreting (and eventually optimizing) methods defining the operational semantics of the language.

Execution context manages the stack and the essential 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 later *executed*.

Memory is responsible for dealing with the actual memory. This includes read/write accesses as well as allocation and garbage collection.

Layouts describes the concrete organization of the internal data of objects.

5. Mate v1: A Complete Iteration over the Methodology

In this section we present Mate v1, a first prototype of a reflective EE that we obtained by following the described methodology. We discuss its main design decisions, mostly concerning the EE-level MOP, and analyze the resulting reflective capabilities.

5.1 Representative Problem

We first selected a practical problem for guiding the design of Mate v1: *unanticipated fine-grained adaptations at run-time*. By this we mean adaptations, at the granularity of objects or methods, that were not anticipated at design time and have to be applied without stopping the system. An example of this kind of scenarios was already presented in section 1. We adopted this problem inspired on [26] that has already pointed out that language-level approaches are suitable alternatives for dealing with fine-grained behavioral adaptations. In addition, we have the hypothesis that many unanticipated scenarios requiring fine-grained adaptations can be properly handled by adapting the EE in a programmatic fashion at run-time. We evaluate this hypothesis in section 6.

5.2 EE-level Metaobject Protocol

Figure 2 presents a sketch of the resulting MOP. The metaclasses are grouped into two main clusters: one concerning the execution and another referring to the organizational aspects of the EE. Compared to the reference architecture it only misses the memory metaclass. We decided to leave the reflective implementation of the memory for future iterations because our case studies do not require reflection for that component. The combination of the capabilities of these metaclasses represent the entire EE-level reflectivity of Mate v1.

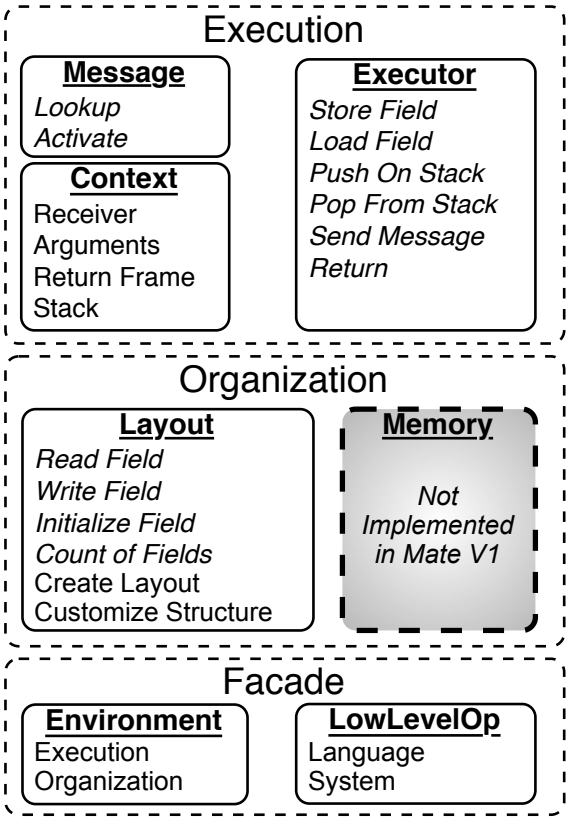


Figure 2: Mate's Metaobject Protocol. The operations highlighted in italics represent behavioral aspects of the EE while the others represent structural aspects.

To determine the reflective capabilities of each metaclass we dealt with requirements that were sometimes in tension between: a) handle the adaptation scenarios. b) explore new reflective capabilities at EE level. c) implement a yet practical EE. What follows is a brief description of the resulting capabilities per metaclass:

- *Message:* Allows developers to redefine (intercede) the method lookup algorithm and the method activation mechanism. In section 6.1.2 we show examples of its application for handling adaptation scenarios.
- *Executor:* Mate v1 implements a bytecode interpreter. This metaclass allows developers to redefine at the language level the behavior of *each* individual bytecode. We only use a subset for the case studies.
- *Execution Context:* Makes it possible to observe and intercede the execution context of each method by interacting with the receiver, the arguments, the caller's context, and the stack. We show an example of its usage in section 6.1.2.

- *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 organization of objects (for the sake of simplicity we omit the details about how objects are organized in Mate v1). Usage examples can be found in section 6.2.

5.2.1 How To Use the MOP

When dealing with structural aspects of an EE entity, reflection is handled by observing and/or altering its corresponding metaobject's fields. For instance, in Mate v1, at instantiation time every object is automatically linked to a *layout* metaobject describing its structure. An *execution context* metaobject is automatically instantiated for every method invocation. Currently, these are the only two metaobjects providing structural reflection at EE level in Mate v1.

In contrast, behavioral intercession is handled by adding methods that overload base-level functionalities. These methods must be incorporated as extensions (via inheritance) of the previously introduced metaclasses: *Message*, *Executor*, and *Layout*. *ExecutionContext* is not included because its operations consider only structural aspects. The subclasses can overload the operations distinguished in figure 2 using italic letters, which are essentially the operations concerning behavioral aspects.

To provide a homogeneous and controlled mechanism for writing these methods, the MOP features two metaclasses: *Environment* and *LowLevelOperation*. They are contained in the *Facade* cluster located at the bottom of figure 2. Environment metaobjects are the only interaction points between the application and the EE metalevel. LowLevelOperation metaobjects only put together instances of the metaclasses with the incorporated method redefinitions. Environments are linked to up to two LowLevelOperation metaobjects: one redefining execution aspects and the other organizational aspects of the EE. Finally, environments can be assigned to entities at different granularity levels: individual objects, set of objects, execution contexts, or even the whole system.

It is worth noting that our mechanism enables only one interceding point for the execution cluster and another for the organization cluster. Although this may appear restrictive, we preferred not to handle the potential complexity of enabling simultaneous (and possibly conflicting) interceding mechanisms in this prototype.

To better illustrate how to use these metaclasses we now present an example. Consider an unanticipated requirement for making a group of objects immutable at run-time like the scenario discussed in the introduction. In addition, there is a need for improving memory consumption by compressing objects that contain several uninitialized fields. Using Mate v1, the operations of the MOP to be redefined are:

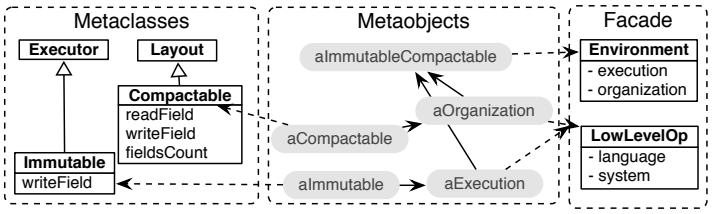
- *Write field* bytecode from *Executor* to throw an exception whenever the system tries to change the value of a field (immutability).
- *Read field*, *write field*, and *field count* from *Layout* to handle the memory compression and make it transparent to the application that the layout has changed.

Figure 3-a shows a possible configuration of metaobjects implementing the aforementioned scenario and 3-b the corresponding steps needed to generate this configuration. To ease readability, we distinguish in the figure between metaclasses, metaobjects and facade metaclasses. Two metaclasses extensions are responsible for the required adaptation: *Immutable* and *Compactable*. *Immutable* extends the *Executor* and *Compactable* the *Layout*. Each overloads the aforementioned operations correspondingly. `aImmutable` and `aCompactable` are instances of these respective metaclasses. `aExecution` and `aOrganization` are the *LowLevelOperation* instances linked to `aImmutable` and `aCompactable` via its instance variables. Note that we need two instances of *LowLevelOperation* because there is a requirement for overloading methods concerning both execution and organizational aspects of the EE. Finally, `aImmutableCompactable` is an *Environment* metaobject linked to `aExecution` and `aOrganization`.

5.3 Analysis of Reflective Capabilities

We present a graphical representation to characterize the reflective capabilities of Mate v1 and compare them with respect to other approaches featuring advanced reflectivity. To the best of our knowledge they are Pinocchio [31] and CLOS [16]. Bellow, we briefly discuss their main differences with Mate v1 while in section 8.2 we provide a more detailed description of both of them.

Figure 4 presents an axis for each EE-level entity appearing in our reference architecture as a means to compare the approaches in a fine-grained manner. We analyze these artifacts estimating the number of reified operations and evaluating their relevance. Note that this ad-hoc evaluation only considers the domain-breadth dimension. Therefore, when solutions have a similar domain-breadth valuation, we would consider the solution with the highest domain-depth as more complete. The top of each axis corresponds to the ideal of full reflection, the base line to having no reflection at all. The dashed wave crossing the figure is an imaginary shape representing the reflectivity that could be achieved in practice. Our long-term goal is to progressively characterize its actual maximum reflectivity value for each of these axis.



a. Configuration of metaobjects for the adaptation scenario

```
1  Subclass Executor and Layout (Immutable, Compactable).
2  Overload required methods on Immutable and Compactable.
3  Instantiate both (aImmutable, aCompactable).
4  Instantiate twice LowLevelOp (aExecution, aOrganization).
5  Assign aImmutable to aExecution.
6  Assign aCompactable to aOrganization.
7  Instantiate Environment (aImmutableCompactable).
8  Assign aExecution to aImmutableCompactable.
9  Assign aOrganization to aImmutableCompactable
```

b. Instantiation of facade objects

Figure 3: How to define and instantiate intercession handlers.

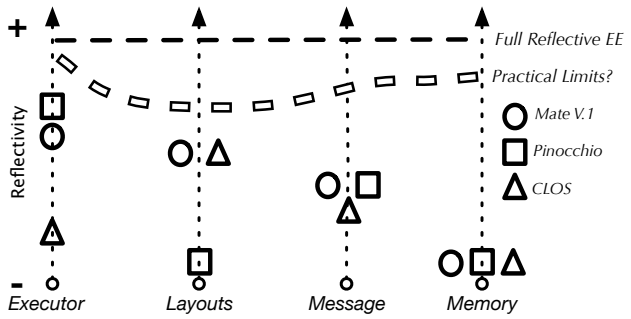


Figure 4: Analysis of reflectivity per component.

Mate v1 reifies the behavior (operational semantics) and part of the structure (execution context) of the executor. We consider this as a higher degree of reflection. However, Pinocchio is above Mate on the executor's axis because beyond this, Pinocchio also enables to work with an unlimited number of metalevels. In contrast, Mate v1 limits the number of metalevels (see next section for a rationale for this). Hence, Pinocchio is more complete in the domain-depth dimension. CLOS reifies several operations on the layout and message components. But even though this enables partial modifications of some program's semantics, it does not include all aspects of the execution. For instance, Mate enables the redefinition of each individual bytecode, and Pinocchio each operation at the AST level. In contrast, CLOS reifies neither of them. Therefore it has lower reflectivity on this axis.

Concerning layouts, Mate v1 provides limited structural reflection on the object formats and intercession handlers for many operations on fields. As a consequence, the reflective capabilities in Mate v1 on layouts can be considered as less powerful than its executor capabilities. Nevertheless, Mate v1 reflective capabilities on layouts outperform Pinocchio's which focuses only on the executor components. Meanwhile, CLOS also features reflective capabilities on layouts by incorporating the concept of *slots* for reifing instance variables. Mate v1 is slightly above CLOS since it supports similar redefinitions for operations on fields, but in addition, Mate v1 also considers a larger part of its structural organization.

Pinocchio and Mate v1 enable to overload the semantics of messages. CLOS has lower reflectivity in this axis because according to our understanding, it does not provide means to intercede the method lookup and activation on individual objects. None of the solutions allow developers to adapt the structure of methods. Finally, as already discussed in section 5.2, Mate v1 does not support reflection for the memory management entity, as neither of the other approaches does.

5.4 Implementation Details

We developed an EE from scratch in order to have full control for tuning and experimenting with advanced reflective capabilities. We adopted Smalltalk as the language to support. The reasons are that Smalltalk fits in the reference architecture, it is well-known for its conceptual simplicity, and it already provides advanced reflective capabilities at the application level. Mate v1 implements an interpreter that supports the complete bytecode of the Cog VM [22], making it compatible with two well known open-source Smalltalk implementations: Squeak [15] and Pharo [5]. It also defines its own object format and a memory manager featuring a mark & sweep garbage collector. In summary, Mate v1 is a research prototype capable of running standard Smalltalk programs that consist of a base-level EE and a MOP reifing its essential components.

5.4.1 Causal Connection

We now present the two different ways of implementing the causal connection between the base and the meta levels of the EE. For each, we discuss completeness and performance as the main obstacles we faced during the implementation of the reflective components.

EE-level Behavioral Reflection

Behavioral reflection is implemented with hooks in the base-level EE. These hooks are usually known as *intercession handlers*. Before executing an EE-level operation that is part of the MOP, like a bytecode, the VM tests

```
1  function executeOperation(op, level) {
2    if (level is Base)
3      if (MOP overloads op)
4        method = MOP.fetch(op)
5        for (metaOp in method)
6          execute(metaOp, Meta)
7      else VM.execute(op);
8    else VM.execute(op);
9  }
```

Figure 5: Mate's intercession handlers implementation. The notation is Java-like pseudo code.

whether there is a metaobject redefining it. If not, the original static implementation of the operation executes. In case the operation is redefined, the VM delegates the responsibility to the corresponding metaobject's method. We now discuss the main limitations of these intercession handlers:

Completeness: We already pointed out the domain-breadth dimension of completeness for each metaclass of the MOP during this section. It is worth adding that in Mate v1 it is not possible to add new hooks (intercession handlers) to the base level at run-time. Therefore, the behavioral reflectivity of the EE cannot be increased on-the-fly. For the domain-depth dimension, Mate v1 allows two levels of execution: *meta* and *base*. Every time the EE delegates the execution of an EE-level operation to the MOP, the level is set to *meta* and no further delegations are possible until the operation returns. This mechanism is illustrated in figure 5 and essentially prevents potential metaregressions by avoiding to dive into further metalevels.
Performance: Each intercession point in the base-level EE requires one extra test whenever the operation is going to be executed. These tests have an impact on the overall performance of the system. However, this kind of mechanism can be optimized using state-of-the-art dynamic compilers (see section 7.3).

EE-level Structural Reflection

We reify the structure of base-level entities using the fields of metaobjects. This mechanism enables a program to observe and change the value of the fields with instantaneous effects. To guarantee the causal connection the base level entity behaves according to the information residing in the corresponding metaobject and vice versa. The difficulties are similar to the previous case:

Completeness: Analogous to the behavioral case, structural reflectivity of the MOP cannot be increased at run-time. Considering the domain-depth dimension, we faced a metaregression issue with layouts. Since layout metaobjects are also first-class objects they must be determined by another layout metaobject. As a compromise (and general) solution, in Mate v1 the structure of metaobjects is determined by fixed layouts.
Performance: The indirections are similar to those of behavioral reflection. However, since the execution of the base level strongly depends on metaobjects, it is unclear how it affects potential code optimizations.

6. Mate v1 Adaptation Capabilities

In this section we present a series of case studies where we analyze how Mate v1 handles *unanticipated fine-grained adaptations* scenarios at *run-time*. Recall, in this scenario the application was not designed for such adaptations and they need to be performed while the system keeps running. Furthermore, they are fine-grained adaptations at the granularity of individual objects or methods. We compare Mate v1 against other adaptation approaches with respect to the feasibility, simplicity and amount of modifications required.

6.1 Immutability

Object immutability [38] is useful, for instance, for software development, testing, and safe updates. For example, during the execution of a test suite it is desirable to enforce that assertion expressions have no side-effects. Activating and deactivating immutability on-the-fly can protect the system against unintended side-effects. Moreover, reference immutability controls mutation at the reference level and enforces more complex mutability properties such as:

- objects being mutable from one reference but immutable from another,
- and propagating immutability through reachable references.

Object and reference immutability have been used to enforce properties such as thread non-interference, parameter non-mutation, and to simplify compiler optimizations [29].

6.1.1 Object Immutability In Mate v1

The following code snippet supplements the general idea already sketched in section 5.2.1 for providing object immutability in Mate v1:

```
1  class Immutable : Executor {
2    function writeField(aNumber, anObject){
3      throw new ImmutableException();
4    }
5  }
6  immutable = new LowLevelOp();
7  immutable.system(new Immutable());
8  immutableEnv = new Environment();
9  immutableEnv.execution(immutable);
10 obj = (new Object()).environment(immutableEnv);
```

On lines 1-5, we subclass the `Executor` metaclass and overload the *writeField* operation so it throws an exception. From line 6 on, the code creates the two required *facade* metaobjects and links them to an instance of the new subclass. The last line installs the *environment* in a new object. Note that deactivating the immutability property requires simply to unset the environment: *obj.environment(null)*.

6.1.2 Reference Immutability In Mate

Smalltalk does not include the concept of references at language level. Hence, for providing reference immutability we extended Mate v1 at run-time for supporting them. We followed Arnaud et al.'s *handles* approach [2]. Handles are like proxies to objects that do not delegate the mutable operations to their targets. For keeping the consistency of Smalltalk, handles must be *transparent*: a user should not be able to distinguish if she is accessing an object directly or through a handle. Moreover, any object accessed through a handle is wrapped into another handle. This way the readonly property *propagates* through the complete chain of objects accessed from an immutable reference.

Our implementation approach of handles in Mate v1 abstracts the semantics of both, the immutability, and the transparency and propagation properties, in two corresponding metaobjects. In the case of immutability we reuse the *Immutable* metaobject from the previous example. For the transparency and propagation we introduce the *DelegationProxy*. DelegationProxy overloads the *method lookup* and *activation* for ensuring the propagation and transparency of handles. Concretely, messages sent to a handle must execute the method from the target object and side-effects must be disabled from that method on in the chain of further activations. Below the code:

```
1  class DelegationProxy : Message {
2    function lookupFrom(aSymbol, aClass) {
3      return super(aSymbol, this.target());
4    }
5    function apply(method) {
6      activation = this.metaActivationObject;
7      activation.fieldAtPut(1,method);
8      activation.fieldAtPut(2,targetOrSelf);
9      activation.fieldAtPut(3,Handle.envForHandles());
10   }
11 }
12 class Handle : Object {
13   function initialize() {
14     this.environment(this.class().envForHandles());
15   }
16   static function envForHandles(){
17     ImmutableReferences = new LowLevelOp();
18     ImmutableReferences.system(
19       new ImmutableExecution()
20     );
21     ImmutableReferences.language(
22       new DelegationProxy()
23     );
24     return new Environment(ImmutableReferences);
25   }
26 }
```

Line 3 delegates the lookup to the superclass implementing the standard algorithm. However, the second parameter ensures that the required method must be looked up in the original object (the target) and not in the handle. The *apply* function overloads how to activate the method. In particular, line 9 selects the environment metaobject to define the semantics within the method context in which the method would be executed. As a consequence, any operation executed in that context is forbidden to modify objects because it is reachable from a readonly reference. Note that this is an example of a metaobject installed at a method activation granularity. Moreover, further calls to other methods inside that context use the aforementioned lookup propagating the same behavior over all the messages that originates from a handle. Summarizing, DelegationProxies ensures the transparency and propagation of handles.

With both semantic elements represented as metaobjects, we can implement handles. In Smalltalk, when an object is created, the method `initialize` is executed. In this case the handles install an environment metaobject to themselves. From line 17 on, we show how this environment metaobject combines the two corresponding metaobjects.

Comparison to other approaches. A classic approach for ensuring object immutability is to instrument the code of every method that may eventually modify the state of immutable objects. Since it requires to modify the application code with non-functional behavior, it is an undesirable solution that increases complexity. Moreover, it is typically limited to the granularity of classes, and then it is not useful for finer-grained scenarios like per-object adaptations. On the other hand, VisualWorks Smalltalk[2] and some Ruby versions use a mutability flag in each object 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, but they require dedicated VM support and do not support propagation.

In the case of reference immutability for dynamic languages, Arnaud's implementation of *handles* [2] requires the duplication of classes. Every class that eventually needs to be immutable has a corresponding *shadow class*. Shadow classes wrap all the methods that change state in order to forbid the modification. This mechanism

[2] http://www.cincomsmalltalk.com

requires changes in the compiler and the instrumentation of bytecodes as a mean to keep classes updated every time a method of the original classes changes. In addition, for ensuring transparency, the approach requires to adapt the VM so that messages sent to the handle are actually dispatched to the methods in the shadow classes. More recently, in [34] an approach based in dynamic proxies modeled handles without requiring modifications to the VM. However, these proxies still require complex code generation for managing method duplications and updates. Furthermore, there is certain lack of transparency: using standard reflection users can identify that they are interacting with proxies.

In contrast to [2] and [34], we showed in this example that with Mate v1 immutability, both in a per-object and per-reference fashion, can be activated at runtime even if the requirement was unanticipated. No ad-hoc VM support is needed and the adaptations do not affect the application-level code. Our solution required only to extend two metaobjects adding no more than 40 lines of code to the system and does not need shadow classes nor method duplications. Eventual modifications to the application would not affect handles since the adaptation semantics are encapsulated in the corresponding metaobjects.

6.2 Changing Object Format

Consider a system which data model includes the representation of people with a considerable amount of optional data. A standard way of modeling this is with a *Person* class that has a field for every piece of information. Smalltalk, like many systems, implement object's fields with an array like representation in contiguous memory addresses. This provides rapid access to them but is considerably inefficient in terms of space whenever most of the fields are uninitialized.

Suppose that Person's instances have twenty fields of which only five are mandatory: name, surname, ssn, address and postal code. Hence, each Person's instance holds twenty contiguous words of memory for its fields although most of them may be empty. For a large application, it can be desirable to reduce memory consumption by compressing the object representation without shutting down the system. One approach is the use of a hash-based representation that ensures that for Person's instance with only mandatory data, only the minimal necessary amount of memory is used. We show below how to exploit Mate's layout capabilities for tackling this scenario at run-time:

```
1  class HashBasedLayout : Layout {
2    function readField (aNumber){
3      index = self fieldIndexForField(aNumber);
4      if (index.isNull()){
5        return null;
6      } else {
7        return self.instVarAt(index);
8      }
9    }
10   function writeField(aNumber, anObject){
11     index = this.fieldIndexForField(aNumber);
12     if (index.isNull()) {
13       throw new NoMoreSpaceException();
14     } else {
15       this.instVarAtPut(index, anObject);
16       this.instVarAtPut(index + 1, aNumber);
17     }
18   }
19   function fieldsCount() {
20     this.class().instanceVariables().size();
21   }
22 }
```

We assume that the required instances were already assigned a layout metaobject describing that the object has only ten fields. The reason is that we implement a hash that uses two fields for each field of Persons. The hash stores the value in the first field and the index of the original field in the second. As a consequence, this layout is only suitable for Persons with only the five mandatory fields filled.

The `HashBasedLayout` metaclass essentially adapts the reading and writing of fields for working with the aforementioned hashed-based organization. For both operations we first need to look for the position on the hash for that field and then do the concrete operation. In addition, for ensuring consistency and transparency, we also redefine the method that returns the quantity of fields of an object. If a user queries the number of fields of a person with the hash-based layout she will still receive twenty as an answer.

Comparison to other approaches. Another approach to avoid the waste of memory caused by optional fields could be the migration of inefficient instances to new classes. This would however require to change both, application code and instantiation points. Furthermore, depending on the implementation, this may require to change several lines of code or the adoption of complex frameworks for managing the migration at run-time. In summary, this alternative increases the complexity by spreading one concept into different classes.

Similar to Mate v1, [32] defines layouts at the language level. But 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. However, this is inefficient when most of the fields are used.

Using Mate v1 it is sufficient to create a new structural *layout* (layouts are first-class) with much less storage consumption than the original one. Complementary, we created the behavioral counterpart of the layout

metaobject and redefine the required operations for being compatible with the structural layout. We managed to adapt the application at run-time adding less than 30 lines of code and saving at least 50% of memory storage for each changed object. We did not need to modify the application code nor add application-level classes to the system. Summarizing, Mate v1 does not depend on how the application is implemented, it does not replicate classes, and it can handle at run-time both, array-based and hash-based storage scenarios.

7. Analysis of Research Questions

In this section we provide (partial) answers to the questions presented in section 3.2. We based the answers on the feedback gathered from the two-phases study presented in the previous sections.

7.1 Feasibility

We think that the degree of reflection reached by Mate v1 is already a good indicator for the feasibility of fully reflective EEs. In addition, the experiments demonstrate how different implementation mechanisms, which were analyzed in section 5.4.1, deal with structural or behavioral aspects of the EE.

7.2 Applicability

Our evaluation focuses on comparing the adaptation capabilities of Mate v1 with existing approaches, considering qualitative aspects such as feasibility and simplicity. We carefully selected examples of adaptation properties that appeared to be of interest in several publications. For completeness and generality, we analyzed cases ranging from behavioral to structural adaptations that need different EE-level adaptations. To the best of our knowledge, we compared Mate to other language-level approaches such as handles. However, currently, there are very few approaches that can address low-level adaptive scenarios at run-time.

While we still need to perform more experimentation, we found initial evidence that fully reflective EEs are a promising approach for handling *unanticipated dynamic fine-grained adaptations*. We demonstrated that with Mate v1 it is possible and straightforward to apply on-the-fly modifications to concrete EE-level functionality without polluting the application model.

7.3 Performance

Building an industrial strength EE capable of handling real life workloads is a complex task that requires considerable engineering resources. While we decided to develop a new EE from scratch (see section 5.4), we focused on studying advanced reflective capabilities instead of common compiler optimizations. Hence, Mate v1 in its current stage is not comparable to industrial EEs in terms of performance.

Nevertheless, recent research shows strong evidence that performance overheads of fully reflective EEs can be minimized. Partial evaluation [37] and meta-tracing [6] frameworks such as Truffle and PyPy [25] have already presented significant speedups for dynamic environments with similar indirection characteristics to Mate v1. Both solutions generate optimized code with guards for ensuring correctness. Furthermore, Marr et al. [19] recently showed that these mechanisms can eliminate the overhead of reflective operations as well as complex metaobject protocols. We showed in section 5.4.1 that our *intercession handlers* pose only one extra level of indirection. As a consequence, we think that Mate v1 fits in the setting of these solutions.

7.4 Abstraction Mismatch

We have not faced with the abstraction mismatch limitation presented in section 3.2 mainly because in Mate v1 we did not implement the lower-level component: the memory manager. In future iterations, we plan to analyze how well the ideas implemented in high-level low-level programming frameworks such as *Benzo* [9] and *org.vmmagic* [12] fit with fully reflective EEs.

8. Related Work

In this section we describe solutions from different domains that are related with Mate v1.

8.1 Models of Reflection

As discussed in section 2.2, Smith's tower of interpreters [27] is widely used for modeling procedural reflection. Compared to our approach, it does not distinguish between different entities at the same abstraction level. Since our goal is to analyze the reflective capabilities of individual EE-level entities and their impact on others, this reflective model is not enough. The denotational semantics of reflection presented by Wand and Friedman [33] presents similar incompatibilities for analyzing reflection in a fine-grained manner.

8.2 Reflective Solutions

Pinocchio first class interpreter [31] is a practical implementation, in the context of an OO language, of the tower of interpreters. The interpreter is first-class and extensible from language level. In contrast to Mate v1, Pinocchio does not impose a fixed number of metalevels for dealing with metaregression but adapts to different levels on demand. On the other hand, Pinocchio is a reflective interpreter while Mate v1 covers more EE-level entities. For instance, Pinocchio is not able to deal with the memory case study of section 6.2, because it does not reify object layouts. Similar to Pinocchio, Asai [3] proposes a first-class interpreter but in the context of a functional language. It shares with Pinocchio the same fundamental differences with Mate.

CLOS [16] 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's main goal is enabling language customizations rather than being a reflective EE, it does not support extensive reflective capabilities for low-level functionalities such as the complete operational semantics of the language. In addition, CLOS is not able to deal with the read-only references of section 6.1.2, in a transparent fashion as Mate v1, because of its limitations for interceding the method lookup and activation on individual objects.

Flexible Object Layouts [32] reifies the internal structure of objects for a Smalltalk environment. Its main reification is the *Slot* that is similar to the Slot of CLOS. Slots can be extended at run-time by redefining four main operations: read, write, initialize and migrate. Mate v1 follows a similar approach for implementing its *Layout* metaobjects.

8.3 Virtual Machines

Several self-hosted approaches for VM construction support some forms of EE-level reflection. Klein [30] for Self has goals similar to Mate but its support for modifying EE-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 [11] 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 Mate, Tachyon was not designed with EE-level reflection as a goal and it does not provide run-time adaptation capabilities of EE-level entities. Maxine [36] for Java, uses abstract and high-level representations of EE-level concepts and consistently exposes them throughout the development process. Development tools like inspectors at multiple abstraction levels provide a live and advanced interactivity with the running VM while debugging. However, Maxine allows to inspect but not to change the EE at run-time. Similarly, in the JikesRVM [1] VM components can be inspected but not modified at run-time. Reflection on VM components is mainly used for the bootstrapping of the system. Mate, on the other hand, focuses on providing interactivity during run-time.

8.4 Dynamic Adaptations

To the best of our knowledge, Partial Behavioral Reflection (PBR) [28] is the most complete reflective solution for supporting unanticipated adaptations. PBR relies on bytecode instrumentation. Hence it is restricted to adaptations regarding only the operational semantics. In addition, instrumentation techniques modify the application code and, from an EE perspective, the original code is not distinguishable from the instrumented code. In contrast, Mate v1 fulfills the adaptations by using reified EE-level components and does not modify the application code. Concretely, Mate v1 focuses on EE-level reflection while PBR depends on application-level reflection for (simulating) the low-level adaptations.

The Iguana/J environment [23] has similar characteristics to PBR. However, Iguana/J provides these capabilities with a MOP that also has similarities with Mate v1 in terms of behavioral adaptive capabilities. Similar to CLOS, Iguana/J provides intercession handlers for method interceptions, reading, and writing of fields. In contrast, Mate v1 allows to intercept a broader set of operations such as the complete operational semantics of the system. In addition, Mate v1 also provides structural EE-level reflective capabilities.

9. Conclusions

In this paper we described our vision of reflective execution environments (EEs) and identified the main research question (RQs) that we want to address. For tackling these RQs we outlined a methodology that propose to iteratively design prototypes with increasing reflective capabilities inspired by the need of handling a set representative case studies. Finally, we evaluated our approach by designing and implementing Mate V1, which introduces reflective capabilities for EE-level concepts through a specialized MOP. Our evaluation also includes a series of case studies for handling unanticipated adaptation scenarios.

We showed that EE-level reflection is feasible and suitable for handling the case studies. We also identified that, in contrast to classic approaches, EEs must be aware of the reifications of themselves in order to allow the language to observe and intercede them. This somehow *lifts* the level of abstraction twice and needs to be taken into account in the development of a reflective EE. We believe the decision of using a MOP to provide EE-level reflection helps reducing this gap. Considering also that we do not yet reify the lower-level component (memory), during the implementation of Mate v1 we did not face obstacles besides the expected classical challenges of completeness and performance.

We consider Mate v1 as a lower bound in the reflectivity space that we are willing to explore. In future iterations, after reifing more components and explor-

ing new reflective capabilities, we expect to encounter new challenges like stronger causal connections, performance issues and other concerns. Some of them may lead to new ways of modeling reflection that may need different techniques than those applied for the traditional application-level reflection. Moreover, a set of quantitative metrics is needed to precisely distinguish the reflectivity of different solutions. Finally, we would also like to experiment with consistency issues when all-encompassing low-level adaptations such as modifying the representation of objects in memory at run-time are possible. Encapsulation and indirection are the standard ways of dealing with them, but other approaches like monitoring of invariants and on-demand compensation might be good alternatives.

Acknowledgments

We would like to thank the reviewers of this paper for their constructive feedback which contributed to improve it. This work was partially supported by the projects UBACYT N° 20020130100384BA, MinCYT (PICT 2012 N° 0724, PICT 2013 N°2341), CONICET (PIP 11220110100596CO, PIP 11220130100688CO), LIA INFINIS and the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS).

References

[1] 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.*, 44(2):399–417, Jan. 2005.

[2] 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*, TOOLS'10, pages 117–136. Springer-Verlag, 2010.

[3] K. Asai. Reflection in direct style. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 97–106. ACM, 2011.

[4] 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*, FoSER '10, pages 17–22. ACM, 2010.

[5] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[6] C. F. Bolz and L. Tratt. The impact of meta-tracing on vm design and implementation. *SCICO*, pages 408–421, Feb. 2015.

[7] 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*, OOPSLA '04, pages 331–344. ACM, 2004.

[8] M. Braux and J. Noyé. Towards partially evaluating reflection in java. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '00, pages 2–11. ACM, 1999.

[9] C. Bruni, S. Ducasse, I. Stasenko, and G. Chari. Benzo: Reflective Glue for Low-level Programming. In *International Workshop on Smalltalk Technologies*, Aug. 2014.

[10] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104. ACM, 2013.

[11] 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*, DLS '11, pages 61–72. ACM, 2011.

[12] 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*, VEE '09, pages 81–90. ACM, 2009.

[13] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. Vmkit: A substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 51–62. ACM, 2010.

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

[15] 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*, OOPSLA '97, pages 318–326. ACM, 1997.

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

[17] P. Maes. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155. ACM, 1987.

[18] J. Malenfant, M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. Reflection '96 Conference, 1996.

[19] 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*, PLDI 2015, pages 545–554. ACM, 2015.

[20] S. McDirmid. Living it up with a live programming language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638. ACM, 2007.

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

[22] E. Miranda. The Cog Smalltalk virtual machine. In *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages*, VMIL '11. ACM, 2011.

[23] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 205–230. Springer-Verlag, 2002.

[24] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 485–495. IEEE Press, 2012.

[25] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953. ACM, 2006.

[26] G. Salvaneschi, C. Ghezzi, and M. Pradella. An analysis of language-level support for self-adaptive software. *TAAS*, 8(2):7, 2013.

[27] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 23–35. ACM, 1984.

[28] 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*, OOPSLA '03, pages 27–46. ACM, 2003.

[29] 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*, OOPSLA '05, pages 211–230. ACM, 2005.

[30] 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*, OOPSLA '05, pages 11–20. ACM, 2005.

[31] 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*, OOPSLA '10, pages 774–789. ACM, 2010.

[32] 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*, OOPSLA '11. ACM, 2011.

[33] 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*, LFP '86, pages 298–307. ACM, 1986.

[34] E. Wernli, O. Nierstrasz, C. Teruel, and S. Ducasse. Delegation proxies: The power of propagation. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 1–12. ACM, 2014.

[35] C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Fine-grained modularity and reuse of virtual machine components. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 203–214. ACM, 2012.

[36] 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.*, 9(4):30:1–30:24, Jan. 2013.

[37] 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 *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 187–204. ACM, 2013.

[38] 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*, ESEC-FSE '07, pages 75–84. ACM, 2007.