Towards Fully Reflective Environments

Onward! 2015

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

Stefan Marr, Stéphane Ducasse
RMoD, INRIA Lille - Nord Europe, France
{stefan.marr, 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 VM development to improve the observability, evolution and adaptability of VMs at run-time. We define the notion of fully reflective execution environments, systems that provide reflection not only at the application level but also at the level of the execution environment (EE). 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.

1. Introduction

With the recently revived 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 programming 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.

A particular example are industrial-strength VMs for Java and JavaScript which are written in low-level static languages, such as C and C++, to meet their 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, or hours, rather than milliseconds, significantly slowing down the development process.

Multiple research projects explored the use of high-level languages for VM construction [1, 11, 29, 35]. 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 also metacircular, *e.g.*, implementing a Java Virtual Machine in Java. «««< HEAD In addition to the use of high-level languages, aspects such as modularity, observability and extensibility have been in the focus as well [13, 34]. However, even the metacircular VM approaches use bootstrapping processes that do not preserve the high-level properties , but instead produce VMs with significantly limited observability and interactivity at run-time. =======

In addition to the use of high-level languages, aspects such as modularity, observability and extensibility have been in the focus as well [13, 34]. However, even the metacircular approaches produce VMs that do not add significant observability and interactivity at run-time. »»»> 8c4b5f4b49d5182698bd2f6bea74b5ec98032552

We advocate 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 very limited nowadays. At the same time, such a ground up interactive design has the potential to also benefit developers administrators, and/or architects by providing them with a high-level interface to interact with the runtime in a uniform way. For instance, application developers can rely on run-time services for adaptability instead of having to develop application-specific solutions, and administrators can use the same interfaces to customize the system based on its utilization at runtime.

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 module to determine the impact of the issue 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 the remainder of this paper we argue that solutions based on fully reflective EEs increase the possibilities to approach such scenarios at language level while also simplifying the solutions. In addition, solutions are more uniform with the existing language infrastructure compared to those that are build ad-hoc on top, *e.g.*, explicitly as part of an application's architecture.

The goal of this paper is start exploring the space of fully reflective EEs. We start 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 a series of unanticipated fine-grained adaptive scenarios concerning low-level aspects. These scenarios include ensuring object and reference immutability, and memory compression. 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 LP 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][2]: a self-hosted prototypical, but functional, reflective EE supporting the Smalltalk programming language and following the reference architecture.
- An empirical validation that demonstrates the feasibility and usefulness of our approach by using the incorporated LP capabilities at the EE level in the context of dynamic adaptation requirements.

2. Background

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

2.1 Reflection

Reflection [26] 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, as well as the ability to write programs within the 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

[1] Mate is a popular infusion in several South American countries.

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

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 related with the degree in which the main concepts of a domain (i.e., components and their responsibilities) are reified. 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 coverage of both the domain-bread and domain-depth aspects of reflection. On the other hand, incorporating more reflection to a system (i.e., making it more complete) increases the flexibility at the cost of affecting its performance.

The domain-depth dimension of completeness is strongly related with the concept of *meta-regression*. It deals with possibly infinite chains of meta-activations. A well-know example from literature that exposes this situation is the tower of interpreters [26]: 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 the subsequent level. In practice, there are different ways to avoid this meta-regression. For instance, one approach is fixing the number of meta-levels so that the top level can not be further reified. Note that this alternative limits domain-depth completeness. All alternatives must face with the same trade-off. 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 them 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 runtime 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 intend 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, in general EEs for dynamic languages are 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:

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

Maxim 2. Uniform Reflective Abstractions:

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

Improving the reflective capabilities of the system with EE-level reflection must not increase the complexity of the programming environment. Developers that work in different domains should be able to focus on enhancing their knowledge on a unique 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 AOP, 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 VM components. This includes analyzing the classical issues, discussed in section 2.2, and also the effects derived from the particular characteristics of the EE domain. To approach this research goal we will intend to answer a series of questions regarding feasibility, performance and applicability.

Starting with feasibility, we would like 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, mainly because of the techniques to avoid the meta-regression problem [18]. This concerns the domain-depth dimension. We are not aware of previous works exploring the reflective capabilities of EEs in a domain-breath fashion. This is particularly interesting in the domain of EEs. The reason is that they include complex elements that are highly coupled and, thus, reflective implementations must ensure causal relationships with more drastic implications. For instance, changes to a GC property or object layout potentially need to be taken into account by a JIT compiler that needs to generate different code.

Taking into account both domain-depth and domain-breadth dimensions of reflection within the particularities of EEs we propose the following 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. [**?**] recently showed that it is possible to remove this overhead in the context of just-in-time compilation. In summary, we are interested in studying also usability and performance aspects of fully reflective EEs:

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 stick to 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 RQs .
5. If there are still unanswered aspects step back to 2.

From now on we start by describing the reference architecture in section 4 and then we proceed with an evaluation organized in two phases. In section 5 we present the first part that consists of the selection of a representative problem and then 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 consisting of an empirical evaluation of the prototype in the context of a set of case studies. This allows us to gather information about applicability and performance 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 an *uniform reflective abstraction* which promotes *separation of the application and EE domains*. Guided by these maxims we decided to design an architecture following these three guidelines:

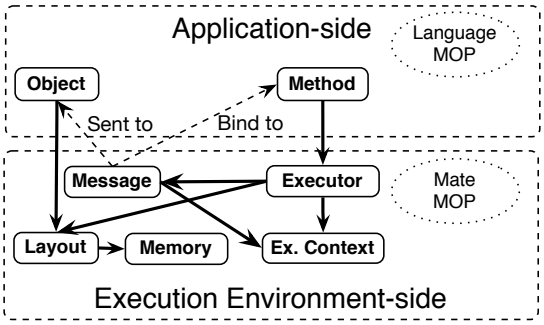


Figure 1: Mate High-Level Architecture.

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 Mirrors' principles.

Figure 1 presents the resulting reference architecture. It forms the basis for the implementation of the successive prototypes. The architecture is divided into two layers: the application and the EE level. 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. However, since our goal is to study in particular EE-level reflection, the language must include a MOP in charge of the application-level reflection. The rounded dashed box at the top right makes explicit this assumption. It is worth noting that most OO solutions implement their reflective architectures with MOPs. In summary, any OO language with an application-level MOP providing reflection is compatible with this architecture.

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 the figure 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 [22], Object-Centric Debugging [23] and Slots [31], that adopt MOPs as a way to deal with low-level concerns.

Finally, MOPs adhering to the Mirrors's principles isolate the reflective capabilities into separate inter-

mediary objects that directly correspond to language structures. The requirement of adhering to these principles at 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 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 of the architecture:

Executor (execution engine) 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.

Messages 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 we have 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 [25] that has already pointed out that language-level approaches are suitable alternatives for dealing with fine-grained behavioral adaptations scenarios. In addition, we had 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 on section 6.

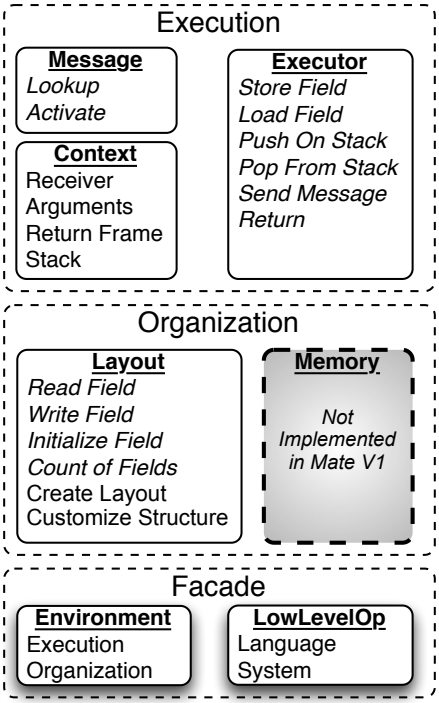


Figure 2: Mate's Metaobject Protocol.

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 reflective capabilities at that level. The combination of these metaclasses and their capabilities represent the entire EE-level reflective capabilities of Mate v1.

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 that can run actual Smalltalk programs. What follows is a brief description of the resulting capabilities per entity:

- *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. The executor 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:* Enables 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 necessity in section 6.1.2.
- *Layout:* Provides means to modify the behaviour of operations interacting with object's fields. Specifically, it enables to redefine their reading, writing, and initialization. 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 Usage

There are two different ways of using the EE-level MOP. Structural reflection of an EE entity 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 metaobject describing its layout. Also, an execution context metaobject is automatically created for every method invocation. Actually, layouts and execution contexts are the only metaobjects providing structural reflection at EE level in Mate v1.

In contrast, behavioral intercession is handled by adding methods. These methods, expressing the new desired behavior, must be incorporated as extensions (via inheritance) of the previously introduced metaclasses: *Message*, *Executor*, and *Layout*. The subclasses can only redefine the set of operations of the MOP concerning behavioral aspects. They are distinguished in Figure 2 with italic letters. Note that *ExecutionContext* is not included in the metaclasses to extend because its operations only consider structural aspects.

«««< HEAD In an attempt to provide an homogeneous and controlled mechanism for writing these intercessory methods, the MOP contains two intermediary metaclasses: *Environments* and *LowLevelOperation*. ======= In an attempt to provide an homogeneous and controlled mechanism for writing these intercessory methods, the MOP features two metaclasses: *Environments* and *LowLevelOperation*. »»»> 8c4b5f4b49d5182698bd2f6bea74b5ec98032552 They are contained in the cluster labeled *Facade*, located at the bottom of Figure 2. «««< HEAD Environment metaobjects aggregate all the EE-level behavioral redefinitions and are the only interaction point between the application and the EE metalevel. This means that every object with its interceded behavior must contain a link to its respective environment metaobject. Actually, environments can be activated at different granularity levels: they can be assigned to objects, set of objects, execution contexts, or even the whole system.

For managing the aggregation , environments are linked to two Low-level Operation metaobjects, one redefining execution and the other organizational aspects. This leaves only one interceding point per cluster. The reason for defining such restricted mechanism is that in this prototype we do not want to handle the complexity of enabling simultaneous (and possible conflicting) interceding mechanisms. We adopted a compromise solution that avoid these problems and yet allowed us to experiment with the case studies. ======= Environment metaobjects are the only interaction point between the application and the EE metalevel. This means that every object with its behavior interceded must contain a link to its respective environment metaobject. Actually, environments can be assigned to entities at different granularity levels: individual objects, set of objects, execution contexts, or even the whole system.

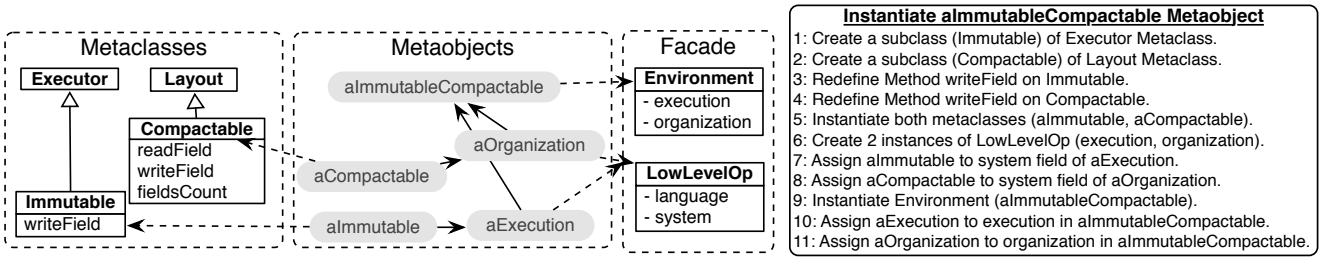
Environments can be linked to up to two Low-level Operation metaobjects: one redefining execution aspects and the other organizational aspects of the EE. This enables only one interceding point per cluster (see Figure 2). This may appear as a restricted mechanism, but in this prototype, we preferred not to handle the potential complexity of enabling simultaneous (and possible conflicting) interceding mechanisms. »»»> 8c4b5f4b49d5182698bd2f6bea74b5ec98032552

We now present an example of how to use the MOP for modifying behavioral and organizational aspects of the EE at run-time. We merge in a use case two common adaptive scenarios from literature: immutability and object's internal data shaping.

Consider an unanticipated requirement for making a group of objects immutable like the one discussed in the introduction. In addition, there is a need for improving memory consumption. The selected approach for coping with this issue is compressing objects that contain several uninitialized fields. Using Mate's MOP, the operations 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 con-



a. Configuration of metaobjects for the adaptation scenario b. Instantiation of facade objects

Figure 3: How to define and instantiate intercession handlers.

figuration. 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 redefining the aforementioned operations correspondingly. `aImmutable` and `aCompactable` metaobjects are instances of these respective classes.

As already mentioned, metaobjects are linked to base level objects using a predefined configuration of facade objects. In the figure this appear as two *Low-level Operations* instances, `aExecution` and `aOrganization`. `aImmutable` and `aCompactable` are linked to these metaobjects via the *language* instance variable. It is worth noting that we need two instances of *Low-level Operations* because there is a requirement for redefining operations concerning both execution and organizational aspects of the EE. Finally, there is an `aImmutableCompactable` object that is an instance of the *Environment* metaclass. `aImmutableCompactable` is linked to the `aExecution` and `aOrganization` metaobjects via its corresponding instance variables.

5.3 Implementation Details

In order to own full control for tuning and experimenting with advanced reflective capabilities we developed an EE from scratch and 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 [21], 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. Even tough Mate v1 is a research prototype, it is still capable of running standard Smalltalk programs. In summary, Mate v1 consist of a static EE and a MOP that reifies its essential components.

5.3.1 Causal Connection

We now present the two different ways of implementing the causal connection between the base and the meta levels. We also discuss about the main (and classical) obstacles we faced during the implementation of the reflective components. Recall from section 3.2 that they are: completeness, performance and metaregression. .

EE-level Behavioral Reflection

```
function executeOperation(op, level)
  if level is Base
    if (MOP overloads op)
      method := MOP.fetch(op)
      for each metaOp in method
        execute(metaOp, Meta)
    else VM.execute(op);
  else VM.execute(op);
```

Figure 4: Mate´s intercession handlers implementation.

Behavioral reflection is implemented with hooks in the static VM also known as *intercession handlers*. While running a standard application's method, the VM is in *baselevel* mode and before executing an EE-level operation, for instance a bytecode implementation, the VM tests whether there is a metaobject redefining it. If not, the original static implementation executes. If the operation is redefined, the VM delegates the responsibility to the corresponding metaobject's method (see Figure 4) and continues execution in *metalevel* mode. On the other hand, while executing EE-level metaobject methods there is no possibility to go further to another metalevel. We now discuss the obstacles:

Completeness: The implementation of this mechanism on Mate v1 is not able to add new hooks on the base level at run-time. Therefore, the MOP behavioral intercessory reflective capabilities can not be extended on-the-fly.

Performance: Each intercession point in the static VM implies one test whenever the operation is going to be executed. All those tests impact on the overall performance of the system. We have the hypothesis that this kind of mechanism is optimizable using state-of-the-art dynamic compilers (see section 7.3).
Metaregression: An operation interceded by a metaobject may be intercepted again by a higher meta level. Mate v1 limits the intercession of operations to only one meta level. This restriction is illustrated in Figure 4. Here we show that Mate only allows two levels of execution: *meta* and *base*. Every time the system delegates the execution of an EE-level operation to the language, the level is set to *meta*. Once in the meta level, no redefinition tests are executed until the meta operation returns and the level of execution is set to *base* again.

EE-level Structural Reflection

We already mentioned that to implement structural reflection we reify the structure of base level entities in metaobjects only with their fields. This mechanism enables 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: Mate v1 is not able to add new metaobjects causally conected to the base level. Therefore, the MOP structural reflective capabilities can not be extended at run-time.
Performance: The indirections are similar to those of behavioral reflection. Since the execution of the base level strongly depends on metaobjects, we are still not sure whether it will favor or jeopardize potential code optimizations.
Metaregression: We faced a metaregression problem with the layout metaobject. Since layouts 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 not determined by another metaobject but they are fix.

5.4 Analysis of Reflective Capabilities

In order to understand where we are and where to go in terms of feasibility, we exhibit in Figure 5 a graphical representation illustrating the reflective capabilities of Mate v1. It also shows how we believe Mate v1 is located with respect to the most advanced approaches that provide reflective capabilities at EE level. According to our best knowledge they are Pinocchio [30] and CLOS [16]. We briefly present in this section their main differences with Mate while in section 8.2 we provide a more detailed description of both of them.

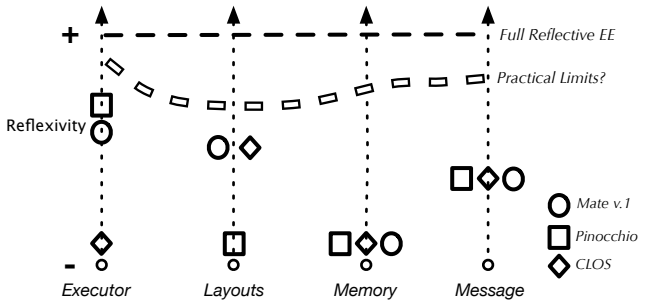


Figure 5: Analysis of reflective capabilities.

Figure 5 has an axis for each EE-level entity as a means to compare the reflective capabilities of the approaches with respect to that entity. We have not found yet a proper way to measure degrees of reflection in a quantitative fashion. As a consequence, for the analysis of these three artifacts we did an approximated measurement which is based mostly on counting the number of refied operations and their corresponding relevance. The top of each axis means full reflection while the bottom no reflection at all. The dashed bold horizontal line at the top represents the ideal EE with each entity being fully reflective. The dashed wave crossing the figure is an imaginary shape representing the practical limits. Our long-term goal is to progressively understand its actual shape.

Mate v1 completely reifies the behavior (operational semantics) and the structure (execution context) of the executor. We consider this as a considerably high degree of reflection. However, Pinocchio is above Mate on this axis because besides reifing the semantics and the execution context it does not limit the number of metalevels. Despite CLOS provides mechanisms to reify operations like assignments or methods, it is from an abstraction level closer to the language and not from an executor point of view.

While all the solutions use intercession handlers to reify the method lookup and activation mechanisms, none feature structural reflection for them. In other words, all enable to redefine the semantics of messages (methods in CLOS) but none allows developers to adapt their structural organization.

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 (lower comparing their height). Nevertheless, Mate v1 reflective capabilities on layouts outperform Pinocchio's which mainly reifies the executor. Meanwhile, CLOS also has powerful reflective capabilities on layouts by incorporating slots for reifing instance vari-

ables behavior. Mate v1 is above CLOS since it supports similar redefinitions for operations on fields but, in addition, Mate v1 also considers the structural organization of fields.

Finally, as already discussed in section 5.2 and similar to the other approaches, Mate v1 does not support reflection for the memory management entity.

6. Mate v1 Adaptation Capabilities

In this section we present a series of case studies where we analyze how Mate v1 handles a set of *unanticipated fine-grained adaptations* scenarios at *run-time*. Recall that this kind of adaptations were not predicted at design time but nevertheless require the system to keep running while are being applied. We compare Mate v1 against other reflective approaches with respect to the feasibility, simplicity and amount of modifications required.

6.1 Immutability

Object immutability [37] is useful 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 is an alternative to protect the system against unintended side-effects. Reference immutability, a step further object immutability, controls mutation at the reference level and enforces more complex mutability properties such as:

- Objects mutable from one reference but immutable from another.
- The propagation degree of the immutability property through reachable references.

Object and reference immutability have been successfully used to enforce, among others, properties such as thread non-interference, parameter non-mutation, and to simplify compiler optimizations [28].

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[3]:

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

In lines 1-5, we subclass the `Executor` metaclass and overload the *writeField* operation so it throws an exception. From line 6 on, we provide a script that creates the two required *facade* metaobjects and link them accordingly to an instance of the new subclass. The last line installs the *environment* in a random base level object. After executing that line, the object becomes immutable. Notice that deactivating the immutability property simply requires to send the message *environment(null)* to the corresponding base-level object.

6.1.2 Reference Immutability In Mate

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

The general idea of our approach to implement handles in Mate v1 is to abstract the semantics of both, the immutability and the transparency and propagation properties, in two corresponding metaobjects. In the case of immutability we can reuse the *Immutable* metaobject that already abstracts that property on the previous example. For the transparency and propagation properties we now 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. Below the code:

```
1  class DelegationProxy : Message {
2    function lookupFrom(aSymbol, aClass) {
3      return super(aSymbol, this.target());
4    }
5
6    function apply(method) {
7      activation = this.metaActivationObject;
8      activation.fieldAtPut(1,method);
9      activation.fieldAtPut(2,targetOrSelf);
10     activation.fieldAtPut(3,Handle.envForHandles());
11   }
12 }
13
```

[3] We provide the code with a syntax similar to those of mainstream OO languages for helping the readers unfamiliar with Smalltalk syntax.

```
14  class Handle : Object {
15    function initialize() {
16      this.environment(this.class().envForHandles());
17    }
18
19    static function envForHandles(){
20      ImmutableReferences := new LowLevelOp();
21      ImmutableReferences.system(new ImmutableExecution());
22      ImmutableReferences.language(new DelegationProxy());
23      return new Environment(ImmutableReferences);
24    }
25  }
```

Line 3 delegates the lookup to the superclass implementing the standard algorithm. However, the required method must be looked for in the original object and not in the handle. That is why the second parameter is the class of the target of the *handle*. In addition, the *apply* function overloads how to activate the method. In particular, line 10 selects the metaobject to rule 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 context 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.

Now that we already have both semantics abstracted in respective metaobjects, we explain how to implement *handles* and assigned to them the required semantics. 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 19 on, we show how this *environment* metaobject combines the two corresponding metaobjects regarding immutability, and transparency and propagation correspondingly.

Comparison of the approaches A classic workaround for ensuring standard 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's functional methods 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[4] and some Ruby versions, already provide a mutability flag to each object for supporting 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

In the case of reference immutability there exist few approaches for dynamic languages [2, 33]. The standard implementation of *handles* reimplements the accessor methods to objects for returning handles. Also other methods for protecting objects from state modifications. These new methods are installed on shadow classes. A shadow class is created for every type of object that eventually needs to be immutable. Managing shadow classes requires compiler changes and bytecode instrumentation for keeping updated whenever methods of the original classes changes. The need for shadow classes is a sign that the underlying infrastructure is not flexible enough to easily support unanticipated scenarios.

In contrast to [2] and [33], we showed in this example that with Mate v1 immutability, both in a per-object and per-reference fashion, can be activated at run-time even if the requirement was unanticipated. No ad-hoc VM support is needed. Moreover, the adaptations do not affect the application level code. Our solution requires only to extend two metaobjects abstracting all the behavior of the immutability, transparency, and propagation properties adding no more than 40 lines of code to the system. Eventual modifications to the application would not affect handles since the adaptation semantics are encapsulated in the corresponding metaobjects. Last but not least, our solution in Mate v1 does not need shadow classes.

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. Suppose also that `Person` instances have twenty fields of which only five are mandatory: name, surname, ssn, address and postal code. Smalltalk, like many systems, represent object's fields via an array like representation with contiguous memory addresses. Hence, each `Person` instance holds twenty contiguous words of memory for its fields. This provides rapid access to them but is considerably inefficient in terms of space whenever most of the fields are uninitialized.

Now suppose that, eventually, due to a peak on memory consumption not approachable by the garbage collector, there is a need for compressing memory without shutting down the system. An alternative for saving memory is a hash-based representation with much less memory slots than fields. In particular, this scheme is profitable for `Person` instances with only their mandatory fields filled. We show below how to exploit Mate's layout capabilities for tackling this scenario at run-time:

```
1  class HashBasedLayout : Layout {
```

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

```
2     function readField (aNumber){
3       index := self fieldIndexForField(aNumber);
4       if (index.isNil()){
5         return null;
6       } else {
7         return self.instVarAt(index);
8       }
9     }
10
11    function writeField(aNumber, anObject){
12      index := this.fieldIndexForField(aNumber);
13      if (index.isNil()) {
14        throw new NoMoreSpaceException();
15      } else {
16        this.instVarAtPut(index, anObject);
17        this.instVarAtPut(index + 1, aNumber);
18      }
19    }
20
21    function fieldsCount() {
22      this.class().instanceVariables().size();
23    }
24  }
```

We assume that previously the required instances were assigned a layout metaobject describing that the object has only ten fields. The implementation of the hash uses two fields for each field of the `Person` class. It stores the value in the first and the index of the original field in the second. This means that this layout is only suitable for instaces with five fields, in this case a Person with only the mandatory fields.

Essentially, the `HashBasedLayout` metaclass adapts the reading and writing of fields for working with this new hashed-based organization. For both operations we first need to look for the position on the hash for that field and then we 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 client 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 One possible workaround to the waste of memory caused by optional fields could be the migration of inefficient instances to new classes. This would require to change both, application's 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, any alternative would increase the application's complexity by spreading one concept into different classes.

Similar to Mate, [31] defines layouts at the language-level. The problem is that this layouts (being at the language level) can be bypassed by primitive operations that do not recognize those constructs. On the other hand, prototype based dynamic languages like Javascript can represent properties of objects with dynamic (hashed-based) dictionaries. However, they may be inefficient when most of the fields are used.

Using Mate we properly adapted the application at run-time. It was enough 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 most important operations for being compatible with the structural layout. The approach required less than 30 lines of code and saves at least 50% of memory storage for each changed object. Moreover, using Mate we neither needed to modify the application's code nor adding new application level classes to the system. Finally, Mate 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 analyze the feedback, gathered from the two-phases study presented in the previous sections, for giving (partial) answers to our research questions

7.1 Feasibility

It is indeed possible to build an EE with extensive reflective capabilities that executes realistic programs. In fact, Mate v1, as the first prototype, can be seen as a lower bound in the reflective capabilities space that we are willing to explore in the subsequent iterations. Moreover, the implementation allowed us to experiment with two different mechanisms depending on if the metaobject Mate v1 structural or behavioral. For each of the mechanisms we were able to analyze the main fundamental obstacles and the impact of the decisions we made for tackling them (see section 5.3.1).

7.2 Applicability

We performed an empirical evaluation focusing mainly in comparing the adaptation capabilities of Mate v1 with existing approaches on qualitative aspects such as feasibility and simplicity. We carefully selected examples of adaptation properties that appeared to be of interest in several publications. For the sake of completeness and generality, we provided cases ranging from behavioral to structural adaptations that need dissimilar EE-side adaptations. We also intended to compare empirically against the most related works according to the best of our knowledge. Unluckily, there are few approaches that can (partially) fulfill low-level adaptive scenarios at run-time. In addition, sometimes it was hard or even impossible to find executable implementations of them. To be as fair as possible, we compared

our prototype to the capabilities claimed at their documentation.

Despite we still need to perform more experimentation, as a general conclusion of this series of experiments we found initial evidence that fully reflective EEs are a promising approach for handling *unanticipated dynamic fine-grained adaptations*. We observed that using Mate v1 it was possible and considerably straightforward to apply on-the-fly modifications to deal with a set of case studies. It was also possible to focus in concrete EE-side functionalities, without ever looking at or polluting the application's functional model. In contrast, other approaches could not implement the scenario, required modifications to the application's functional code (*e.g.*, whole program instrumentation), or even recompilations of the EE.

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 5.3), we focus 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 works show strong evidence supporting that performance overheads of fully reflective EEs might be considerably mitigated. Modern Partial Evaluation [36] and Metatracing [6] frameworks like Truffle and PyPy [24] have already presented significant speedups for dynamic environments with similar indirection characteristics to Mate v1. Both solutions generate optimized code with guards that are checked everytime the code is going to be executed for ensuring correctness. Furthermore, Marr et al. [**?**] recently showed that it is possible to even eliminate the overhead of several language-side reflective operations using speculative optimization techniques. We showed in section 5.3.1 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 studied the eventual abstraction mismatch limitation presented in section 3.2 mainly because we have not (yet) faced with this problem. The main reason is that 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.

7.5 Discussion

We finally want to discuss about the process of implementing reflection at the EE-level and compare it to the classic approaches. In classic approaches, the application-level is executed by a EE that gives support to the language's reflective capabilities. One interesting aspect of EE-level reflection is that EE must be aware of the reification of itself in order to allow the language to observe and intercede it. This somehow *litfs* the level of abstraction twice and needs to be taken into account in the development of the EE and its MOP.

The design decision of using of an uniform MOP at the application level went in the direction of making the development similar to the classical approach. In addition, we tried to mitigate the complexity by defining a minimal architecture that identifies only the essential components. As a consequence, during the implementation of Mate v1 we mainly faced with the classical obstacles of reflective implementations. We showed how we dealt with them in section 5.3.1.

In future iterations, after reifing more components and try with new reflecitve capabilities, we expect to encounter new challenges in how to deal with stronger causal connections, performance issues and other concerns. Some of them may lead to new ways of modeling reflection that may differ from the traditional techniques applied to provide reflection at the application level.

8. Related Work

In this section we describe solutions from a broad range of domains that are related with Mate v1.

8.1 Models of Reflection

Smith introduced the notion of the tower of interpreters where each level of the tower is responsible for interpreting the lower level [26]. The lowest level, *i.e.*, the base level, is the application. This simple model was widely used for modeling procedural reflection. Unluckily, the tower of interpreter does not distinguish between different entities at the same abstraction level such as the memory manager or the layouts. It is coarse-grained. Since we want to analyze the reflective capabilities of individual EE-side entities and their impact on others, the reflective tower does not fit with our goals. Moreover, the denotational semantics of reflection presented by Wand and Friedman [32] presents similar incompatibilities for analyzing reflection in a fine-grained manner.

8.2 Reflective Solutions

Pinocchio first class interpreter [30] 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-side. In contrast to Mate,

Pinocchio does not impose a fix number of metalevels. It adapts to different levels on demand. On the other hand, Pinocchio is only a reflective interpreter while Mate covers a more wide range of EE-side entities. Concretely, Pinocchio is not able to deal with our second case study that deals with object layouts. Similar to Pinocchio, Asai [3] proposes a first-class interpreter but in the context of a functional language. It shares with Pinnochio the same fundamental differences with Mate.

Flexible Object Layouts [31] reify the internal structure of objects for a Smalltalk environment. Its main reification is the *Slot*, a language level representation of an instance variable (field). Similar to the MOP of Mate v1, Slots can be extended at run-time by redefining four main operations: read, write, initialize and migrate. Mate follows a similar approach for implementing its *Layout* metaobjects.

Actually, *Slots* were first introduced by the Common Lisp Object System (CLOS) [16]. CLOS 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. As a consequence, CLOS MOP inspires the development of several future MOPs. However, CLOS main goal is to provide a customizable language, not a reflective EE. Therefore, it does not provide extensive reflective capabilities for low-level functionalities such as the operational semantics of the language. Run-time adaptations are only expressible in terms of extending the semantics of method applications and a set of operations on slots. In contrast, Mate's main goal is to provide a reflective EE, and as a consequence, extensions are also expressible in terms of lower-level components of the EE such as each individual bytecodes.

8.3 Virtual Machines

Several self-hosted approaches for high-level VM support some forms of EE-side reflection. Klein [29] for Self have similar goals to Mate but its support for modifying EE-side 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.On the other hand, 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-side reflection as a goal and it does not provide run-time adaptation capabilities of EE-side entities. Maxine [35] for Java, uses abstract and high-level representations of EE-side concepts and consistently exposes them throughout the development process. Development tools like inspectors at multiple abstraction levels provide a live and advanced interaction with the running VM while debugging. However, Maxine enables to introspect, but not to intercede the EE at run-time. Similarly, Jikes [1] RVM does not expose the VM components (written in Java) at run-time. Reflection of VM components is mainly exploited for the bootstrapping of the system and so it is considered as compile-time reflection. Mate, on the other hand, focuses on providing a live interaction during run-time.

8.4 Dynamic Adaptations

Partial Behavioral Reflection (PBR) [27] is the most complete reflective solution to the best of our knowledge for supporting unanticipated adaptations. Since PBR is by design based on bytecode instrumentation, in contrast to Mate v1 it is restricted to adaptations of only the operational semantics. Moreover, instrumentation techniques pose limitations that potentially impede their applicability such as eventual unexpected behaviors when original source code is not distinguishable from the instrumentation code. In contrast, Mate v1 fulfill the adaptations by using reified EE-side behavior without interfering with the application's model. Essentially, the main difference is that Mate v1 provides EE-side reflection while PBR depends on application-side reflection for (simulating) the low-level adaptations.

The Iguana/J environment [22] 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. The main difference is that the MOP of Iguana/J, similar to CLOS, only 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-side reflective capabilities.

9. Conclusions

We described our vision on reflective EEs and identified the main research question that we want to address. We presented a methodology to tackle those RQs and designed Mate v1, a first prototype that we characterized and evaluated on a series of case studies. From the evaluation we conclude that EE-side reflection is suitable for handling unanticipated adaptation scenarios. Moreover, the degree of reification and reflectivity reached with Mate v1 is already a good indicator for the feasibility of fully reflective EEs. However, future work needs to go further to precisely define the fundamental and practical limits of reflective EEs.

In order to do so we will further study reflective capabilities at EE-side by developing new prototypes that explore different degrees of reflection. Also a set

of quantitative metrics is still needed to more precisely distinguish them. In addition, the performance impact and plausible optimizations need to be studied more in depth. While approaches like [19] seem to be applicable to Mate, it still needs to be shown that a fully reflective EEs can reach a performance similar to those of classic VMs. In this context, it is also interesting to study how this idea could be gradually applied to mainstream VMs.

Finally, we want to understand which are the best mechanisms for ensuring the consistency after low-level changes, like the representation of objects in memory, on a live EE. Encapsulation and indirection are the standard ways of dealing with this issue, but other approaches like monitoring of invariants and on-demand compensation might be good alternatives.

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. TOOLS'10. Springer-Verlag.

[3] K. Asai. Reflection in direct style. In *GPCE '11*, pages 97–106, New York, NY, USA, 2011. ACM.

[4] L. Baresi and C. Ghezzi. The disappearing boundary between development-time and run-time. 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. *Science of Computer Programming*, 2013.

[7] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04*. ACM.

[8] M. Braux and J. Noyé. Towards partially evaluating reflection in java. *SIGPLAN*, 34(11):2–11, Nov. 1999.

[9] C. Bruni, S. Ducasse, I. Stasenko, and G. Chari. Benzo: Reflective Glue for Low-level Programming. In *IWST'14*, Cambridge, United Kingdom, 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. *SIGPLAN Not.*, 48(6):95–104, June 2013.

[11] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: an experience report. DLS '11, pages 61–72, New York, NY, USA, 2011. ACM.

[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. 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 *VEE '10*, pages 51–62, 2010.

[14] M. Haupt, C. Gibbs, B. Adams, S. Timbermont, Y. Coady, and R. Hirschfeld. Disentangling virtual machine architecture. *Software, IET*, 3(3):201–218, 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 *OOPSLA'97*, pages 318–326. ACM Press, Nov. 1997.

[16] G. Kiczales. *The art of the metaobject protocol*. MIT press, 1991.

[17] P. Maes. Concepts and experiments in computational reflection. OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM.

[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 *PLDI '15*, PLDI '15. ACM, 2015. (to appear).

[20] S. McDirmid. Living it up with a live programming language. In *OOPSLA '07*, pages 623–638.

[21] E. Miranda. The Cog Smalltalk virtual machine. In *VMIL '11*. ACM, 2011.

[22] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *ECOOP 2002*, pages 205–230. Springer, 2002.

[23] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *ICSE '12*, pages 485–495, Piscataway, NJ, USA, 2012. IEEE Press.

[24] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[25] G. Salvaneschi, C. Ghezzi, and M. Pradella. An analysis of language-level support for self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(2):7, 2013.

[26] B. C. Smith. Reflection and semantics in lisp. POPL '84, pages 23–35, New York, NY, USA, 1984. ACM.

[27] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. OOPSLA '03. ACM, 2003.

[28] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to java. OOPSLA '05, pages 211–230, New York, NY, USA, 2005. ACM.

[29] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory pro-

gramming environment. In *OOPSLA '05*, pages 11–20. ACM.

[30] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, and O. Niestrasz. Pinocchio: Bringing reflection to life with first-class interpreters. OOPSLA '10. ACM.

[31] T. Verwaest, C. Bruni, M. Lungu, and O. Nierstrasz. Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. OOPSLA '11, pages 959–972. ACM, 2011.

[32] 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, New York, NY, USA, 1986. ACM.

[33] E. Wernli, O. Nierstrasz, C. Teruel, and S. Ducasse. Delegation proxies: The power of propagation. MODULARITY '14, pages 1–12. ACM, 2014.

[34] C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Fine-grained modularity and reuse of virtual machine components. In *AOSD '12*, pages 203–214, 2012.

[35] 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.

[36] 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! '13*, pages 187–204, New York, USA, Oct. 2013. ACM.

[37] Y. Zibin, A. Potanin, M. Ali, S. Artzi, and M. D. Ernst. Object and reference immutability using java generics. In *ESEC/FSE*, pages 75–84. ACM, 2007.