Metaphysics: Towards a Robust Framework for Remotely Working with Potentially Broken Objects and Runtimes

Javier Pimás
Palantir Solutions
Buenos Aires, Argentina
javierpimas@gmail.com

Stefan Marr
Johannes Kepler University
Linz, Austria
stefan.marr@jku.at

Abstract

Dynamic Metacircular Runtimes (DMRs) enable a new way of developing language virtual machines (VMs). Instead of writing VMs by manipulating files, DMR programmers modify methods, classes, and generally objects of a running system. However, while this approach allows us to understand the behavior of a VM more easily by seeing it *alive*, it is also problematic, because the development environment relies on the VM to be stable and work correctly, but even simple changes could break the whole VM.

In this work, we experiment with adapting live programming tools to make them safer for the development of core DMR components. We make them robust so that they can work on DMRs that crashed or are not fully working. This paper describes *Metaphysics*, a framework that combines mirrors and proxies to reify different message execution semantics, allowing execution of code by mixing behavior of a target, possibly broken DMR with an IDE DMR that is fully working. With Metaphysics we built native code debugging and profiling tools that use of the metacircularity of our Bee DMR. They enable the dynamic, fast-paced *edit-test* cycle that we are used to from developing application-level code, which is a major improvement over the classic edit-compile-get-coffee-test cycle used for state-of-the-art VMs.

CCS Concepts • **Software and its engineering → Object oriented languages**; **Runtime environments**; *Garbage collection*; *Dynamic compilers*;

Keywords debugging, remote, dynamic, metacircular, runtimes

ACM Reference Format:
Javier Pimás and Stefan Marr. 2017. Metaphysics: Towards a Robust Framework for Remotely Working with Potentially Broken Objects and Runtimes. Presented at *Workshop on Meta-Programming Techniques and Reflection (Meta'17)*. 5 pages.

1 Introduction

Dynamic Metacircular Runtimes (DMRs) [7] are meant to improve developer understanding of language virtual machines (VMs) and simplify the modification of VM components. DMRs expose vital components of the VM to application developers via the standard tooling, e.g., inspectors, debuggers and code browsers, which run within the same VM. These components can be changed quickly, giving instant feedback. However, even small changes to vital components can lead to the crash of the entire VM leaving high-level tools unresponsive.

The main problem is that current high-level tooling does not interact well with low-level VM aspects, especially when VMs reached an inconsistent state, perhaps with corrupted memory, or simply crashed. In this work, we investigate how this can be addressed.

Our work is based on the Bee DMR [7], a metacircular Smalltalk implementation with native code compilation and garbage collection. Bee's performance is roughly similar to the CogVM's, a Smalltalk VM used by Squeak and Pharo. In the first stages of Bee's development, we used standard low-level tools such as GDB and IDA Pro for debugging low-level issues. While they are customizable and could be adapted to use Bee's meta information, they do not provide the interactivity expected by Smalltalk developers. Furthermore, customizing these tools means duplicating the mechanisms to deal with meta data that exist in the DMR. Changes to the format of meta data require changes to external tools too. To avoid these issues, we created a new set of tools that enrich the programming environment. Unlike the set of existing Smalltalk tools such as inspectors, browsers and debuggers, the new tools are designed to interact with a different Bee VM process, allowing developers to work with objects in the target VM which may be paused for debugging. The process might have been paused by the operating system because it executed an invalid operation, or by the developer for debugging purposes. Either way, the process is still alive, its memory can be read or written, and it may even be possible to change it and to let it continue executing.

For our new tools we developed the *Metaphysics* framework, inspired by ideas of Bracha and Ungar [1]. This framework allows access to remote objects by providing mirrors for structural reflection and proxies for behavioral intercession, making possible to adapt message sending, i.e., method invocation semantics, based on the specific need of the current situation.

This paper describes the problems that motivated the creation of Metaphysics and the object model we are working on to solve these problems.

2 Dealing with Objects in Different Contexts

This research is guided by our desire to work on a *target* system that potentially is not working. This could be either a process that was paused by the operating system or a memory dump of a program that crashed.[1] Either way, no execution is taking place in the target system. We can access such a system via APIs that lets us read and write memory, registers, processor flags, etc.

A DMR developer is assumed to use a fully-working VM for the tools, i.e., IDE. This VM is expected to be similar to the target system, but it does not need to be identical. There might be different classes loaded in each system, or methods of the same classes could differ. Using a similar system ensures that information missing about the target system, for example the shape of a class, can be supplemented with information that is expected to be equivalent.

The contexts in which DMR developers use these tools vary greatly and there does not seem to be a single approach that matches all situations. Some tasks require only access to structures, others require execution of code in the target system, and in other cases one might need to simulate what a method activation would do in the target system. Furthermore, the required execution semantics can change dynamically as the exploratory development tasks evolve.

The reminder of this section gives examples for the required execution semantics that we came across while working on the Bee Smalltalk DMR.

2.1 Remote Object Discovery

As first example, let us consider a situation in which we need to identify objects in the target VM. Let us assume a VM gets paused and we want to obtain information about its most important objects: its addresses, classes, globals, symbols (identifiers in Smalltalk), etc. We have to discover where objects are and we do not have yet access to the symbols of the system, so it may not be possible to send messages to objects. The memory of the target system contains all needed metainformation, e.g., the dictionary of globals, from where we can fetch objects such as true, false and nil, Object class, its metaclass, CompiledMethod, etc. Given the address of an object it is possible through meta information to access its fields and internal structure. This situation is a good candidate to be solved with mirrors [1], because it requires mainly structural reflection.

[1]While we expect to be able to work with crash dumps in the future, we have not worked on this yet.

2.2 Remote Code Execution

When profiling a system, we want to gather information about the execution trace by looking at the compiled methods in the stack. While the situation is similar to the previous example, it poses new challenges. A Smalltalk stack can be traversed with mirrors to gather a call trace. However, obtaining the source code of compiled methods is more complex, because Bee's mirrors do not support it directly. Accessing the internal structure of a compiled method to find its source code directly would be a bad practice, because it means violating encapsulation.

In general, objects are designed in ways that decouple the external view from the internal structure. In practice, this means when accessing an object's internal structure from the outside, it can appear very different from the actual machine representation. Most of the time, the provided view is more high-level and we would prefer to work with the high-level view through an object's external interface, and only go into the internals when it is really necessary.

Back to our example of fetching the sources of a compiled method, a method such as #sourceCode encapsulates this process, which can be complex. Therefore, we want to execute it in the target system to retrieve the result.

2.3 Simulation of Remote Code Execution

When a system is paused it may have been working up to this point or may have crashed (in that case it is still alive and we have trapped the exception by externally attaching to the process). We might want to inspect objects of such a system, understand the execution context, send messages to objects, and perhaps make changes to recover from a crash. However, the objects we are accessing could be broken, which means that their own memory or the memory that determines their behavior (containing class, methods, etc) could be corrupted.

Let us consider again the example of fetching the sources of a method, this time in a crashed system. In this case, we may not be able to remotely execute #sourceCode in the target system because it could modify state of the runtime and make debugging more difficult. As an alternative, we could take the code of #sourceCode from the target system and simulate its execution by interpreting it locally. Of course, obtaining the source code of the remote #sourceCode method means we need the source of #sourceCode causing an infinite recursion. This can be solved by bootstrapping an initial version of this method from the local system, use it to simulate execution on the target system and continue from there.

Our intention is then to create a sufficiently generic framework which lets the developers handle all the situations presented naturally, and that can change over time in an interactive debugging session to adapt to most needs possible.

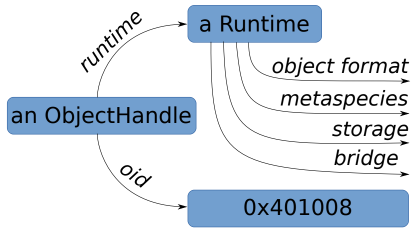

Figure 1. Object handles are opaque references to objects in a target runtime. They are able to answer querys about the objects they point to only by delegating them to their runtime.

3 The Design of the Metaphysics Framework

This section discusses how the Metaphysics framework solves the problems identified in the previous section for the Bee DMR. The Bee runtime is written in Smalltalk, using the standard Smalltalk browsers, inspectors, and debugger. It is a self-hosted DMR [7]. Its structure, as well as Bee's object format and ABI are detailed in Pimás et al. [6, sec. 3].

3.1 Base Metaphysics Concepts

The framework builds on the reification of a Smalltalk runtime and its objects. It is implemented using the underlying OS API for external process debugging to enable communication with the objects of the target system. The main concepts of the Metaphysics framework are depicted in figure 1.

A Handle represents an entity inside a given runtime. A Runtime models the environment in which particular entities pointed to by handles are stored, and is configured to provide low-level meta information of such handles via the following fields:

- a Storage, that abstracts the OS API to read and write from the target process memory,
- an ObjectFormat, that understands and is able to read and write object headers in the target system,
- a Metaspecies, that knows the shape of classes in the target system, allowing to read and write slots of objects in the target system without using their metadescription,[2]
- a Bridge, that is able to locate and deliver handles for global objects in the target system.

There are two kinds of handles: ObjectHandles for referring to typical objects in the runtime, and FrameHandles to refer to stack frames. ObjectHandles contain an oid, which is the address of an object in the target runtime. FrameHandles contain a slot which represents a stack frame in its associated runtime. Handles are usually short lived, used and discarded while the target runtime is paused. Therefore, we do not need to update them when the GC of the target runtime moves objects.

[2]Metaspecies are needed at initial stages, to provide early access to the metalevel of objects and for raw access to object slots using mirrors.

3.2 Mirrors

Structural reflection on the objects of the target runtime is achieved with mirrors. A Mirror in our Metaphysics framework is no more than a container for a Handle. Different subclasses provide an interface that allows us to perform basic queries on the objects mirrored. An ObjectMirror allows getting an object's size and slots, and it can also provide a mirror on the object's class.

In Bee, mirrors work as a layer that makes a distinction between objects of the target system and local ones. By default, methods that access object internals do not do any copying, but return new mirrors that contain the direct references pointed to by the slots of objects. Consider the following example:

```
ClassMirror>>#name
  | name |
  name := self getInstVarNamed: #name.
  ^name asStringMirror

StringMirror>>#asLocalString
  ^handle asLocalString

ObjectHandle>>#asLocalString
  ^runtime stringOf: oid

Runtime>>#stringOf: oid
  | size |
  size := objectFormat sizeOf: oid.
  ^storage stringAt: oid sized: size
```

When a class mirror is sent the message #name, the result is a mirror to the string stored in the name slot of that class. To do something meaningful with this mirror, it normally needs to get a local copy of the string, which is done through the method #asLocalString.

The mirror model of our framework, in conjunction with the design presented in section 3.1 is enough to solve the problem of object discovery in the target system stated in 2.1.

3.3 Subjects, Gates and Execution Semantics

For more complex behavior, a model that allows varying message execution semantics was created. The experience we obtained from the scenarios described in sections 2.2 and 2.3 showed that mirrors were only a first approximation to the dynamic environment we desire to work in. Mirrors allow us to obtain references to objects of the target system by reflecting on their internal structure. Afterwards, given a reference to an object, we want to treat it as if it were a local object, but giving arbitrary semantics to the messages sent to it. This requires another mechanism that realizes

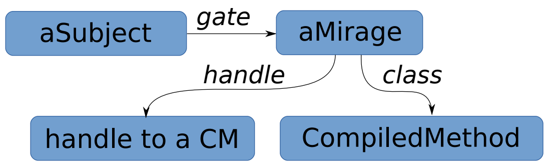

Figure 2. When sent a message, aSubject delegates execution to aMirage, which in turn simulates the execution of the message. In this case the proxied object is a compiled method, so the mirage will create an interpreter for the local CompiledMethod class. The interpreter will perform lookup on that class and traverse the AST of the method found to generate a result.

the different possible execution semantics. To tackle these varying needs we designed the notion of *subjects*.

A Subject is a proxy to an object in the target system. It only understands the #doesNotUnderstand: message, which it overrides to delegate execution to *gates*. Gates of different types implement different execution semantics, working as a strategy pattern.

```
Subject>>#doesNotUnderstand: aMessage
  ^gate dispatch: aMessage
```

The only slot of a subject is its gate, and in turn the gate points to the object handle. By overriding the #dispatch: method, gates can realize arbitrary behavior on the subject when receiving a message. Subjects and gates are separated to keep the subject interface minimal and that most messages are intercepted by the #doesNotUnderstand: handler.

During Bee development we identified three kinds of gates, which we can choose depending on the situation:

Triggers cause execution of the message on the target process, by modifying the process state, resuming the process and returning a result. They correspond directly to the semantics needed to tackle problems described in section 2.2

Direct gates cause the local interpretation of the message, fetching the source code of methods from the object in the target system to which the message was sent.

Mirage gates cause the local interpretation of message, fetching the source code of methods from a local class that is equivalent to the one of the object in the target system.

Direct and Mirage gates correspond to the different kinds of execution semantics desired in section 2.3. They required the implementation of a source code interpreter. This interpreter takes as input an AST, a receiver and an array of arguments. It iterates the nodes of the tree, evaluating them and finally returning the result of the evaluation.

Figure 2 depicts the collaboration of different objects of the framework.

Local simulation of message execution for an object of the target system returns a handle to another object. Normally, that resulting object is referred by the receiver of the message and is also stored in the target system. In that case the runtime of the receiver is assigned to the returned handle. In other situations, simulation can result in handles for local objects. For example, when a new object needs to be created during interpretation, it is instantiated locally by the interpreter, which returns an object handle pointing to the new object. The runtime assigned to the new handle is one that represents the local system. In that case, the oid of the handle is an actual pointer to a local object.

To access an object via different semantics the developer usually takes a handle to such an object and crates a new subject with the desired kind of associated gate. Messages with arguments can be sent to subjects, as long as each argument is also a subject.

4 Related Work

As previously mentioned, we built on the ideas of mirrors [1]. Similar work includes for instance Mirages [4], which try to reconcile mirrors with behavioral intercession in AmbientTalk, an actor-based distributed OO language.

The general notion of remote debugging of OO environments has been studied, too. The low-level IPC mechanism used by Metaphysics is similar to the one used by Maxine Inspector [3] and Jikes RDB[2]. Furthermore, Maxine Inspector uses mirrors to access remote objects. Another approach is taken in Mercury [5], which implements remote debugging using reflection via mirrors and other kind of middleware. While it provides similar functionality to Metaphysics, its mechanisms differ widely. On one hand, Mercury introduces a modified language, *MetaTalk*, where the metalevel is structurally decomposed (via stateful mirrors), and the target system has to run a modified VM that is able to exploit the language features; it provides an adaptable middleware, *Seamless*, to communicate both systems, a runtime-debugging support layer has to be embedded to the target environment, and reflection support is limited to components wrapped by mirrors. On the other hand, Metaphysics uses the well-known Smalltalk-80 metamodel, and the target environment is run unmodified, without middleware or any special debugging support layer. Reflection on the system is based on the already existing reflective facilities of both the target and the host system. Metaphysics was thought for debugging remote processes where the communication to the remote environment is trusted and fast, like a remote process on the same machine.

Salkeld and Kiczales [10] propose holographic objects to deal with snapshots of crashed programs, while Polito et al.

[9] and Polito [8] propose a virtualization infrastructure for OO high-level language runtimes.

5 Conclusion

We presented Metaphysics, a framework for accessing objects with arbitrary message execution semantics. We have developed this model to improve the workflow of system programmers in our Bee DMR. This framework is currently used in our own dynamic native code debugger and profiler. These tools make use of the metalevel information present in the target Smalltalk image instead of using debug information formats such as DWARF or PDBs, and can be adapted arbitrarily at run time while we attach to a process, a feature we already took advantage of and that we expect to keep using.

Acknowledgments

The authors want to thank Gerardo Richarte, Javier Burroni, Valeria Murgia, Leandro Caniglia, Jan Vrany and the development team of Palantir Solutions for providing valuable ideas, discussions and reviews, and being in charge of the development and maintenance of all Bee libraries. This work was funded by Palantir Solutions. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

References

[1] Gilad Bracha and David Ungar. 2004. 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 2004, October 24-28, 2004, Vancouver, BC, Canada.* 331–344.

[2] Dmitri Makarov and Matthias Hauswirth. 2013. Jikes RDB: a debugger for the Jikes RVM. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13).* ACM, 169–172.

[3] Bernd Mathiske. 2008. The Maxine Virtual Machine and Inspector. In *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA Companion '08).* ACM, 739–740.

[4] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. 2007. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the 2007 symposium on Dynamic languages.* ACM, 89–100.

[5] Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, and Marcus Denker. 2015. Mercury: Properties and design of a remote debugging solution using reflection. *The Journal of Object Technology* 14, 2 (2015), 36.

[6] Javier Pimás, Javier Burroni, and Gerardo Richarte. 2014. Design and implementation of Bee Smalltalk runtime. (2014).

[7] Javier Pimás, Javier Burroni, Jean Baptiste Arnaud, and Stefan Marr. 2017. Garbage Collection and Efficiency in Dynamic Metacircular Runtimes. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17).* ACM, 12.

[8] Guillermo Polito. 2015. *Virtualization Support for Application Runtime Specialization and Extension.* Ph.D. Dissertation. Universit e des Sciences et Technologies de Lille.

[9] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. 2013. Virtual smalltalk images: Model and applications. In *21th International Smalltalk Conference-2013.* 11–26.

[10] Robin Salkeld and Gregor Kiczales. 2013. Interacting with dead objects. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 203–216.