End-User Abstractions for Meta-Control: Reifying the Reflectogram

N. Papoulias[a,b], M. Denker[c], S. Ducasse[c], L. Fabresse[d]

[a]*IRD, UMI 209, UMMISCO, IRD France Nord*
http:// www.ird.fr/
[b]*Sorbonne Universites UPMC, Univ. Paris 06, France*
http:// upmc.fr/
[c]*RMoD, Inria Lille Nord Europe*
http:// rmod.lille.inria.fr
[d]*Mines Telecom Institute, Douai*
http:// www2.mines-douai.fr

Abstract

Reflective facilities in OO languages are used both for implementing language extensions (such as AOP frameworks) and for supporting new programming tools and methodologies (such as object-centric debugging and message-based profiling). Yet controlling the runtime behavior of these reflective facilities introduces several challenges, such as computational overhead, the possibility of meta-recursion and an unclean separation of concerns between base and meta-level. In this paper we present five dimensions of meta-level control from related literature that try to remedy these problems. These dimensions are namely: temporal and spatial control, placement control, level control and identity control. We then discuss how these dimensions interact with language semantics in class-based OO languages in terms of: scoping, inheritance and first-class entities. We argue that the reification of the descriptive notion of *reflectogram* can unify the control of meta-level execution in all these five dimensions while expressing properly the underlying language semantics. We present an extended model for the reification of the reflectogram based on our additional analysis and validate our approach through a new prototype implementation that relies on byte-code instrumentation. Finally, we illustrate our approach through a case study on runtime tracing.

Keywords:
Reflection, Intercession, Reflectogram, Explicit Control

1. Introduction

The notion of reflection was formally introduced to programming language literature by Brian Cantwell Smith in 1982 (by means of the programming language 3-LISP [26]). In OO reflective systems, reflection is concretized using a MOP (*Meta-Object Protocol*) [11]. A meta-object is a regular object that describes, reflects or defines the behavior of a notion of the language in question [13]. The process of materializing a notion of a language (such as an object, a class, a context or a method) as an object inside the language itself is called *reification*. Reflective facilities in OO languages [5] are used both for implementing language extensions such as AOP frameworks [29] and for supporting new programming tools and methodologies such as object-centric debugging [23] and message-based profiling [1].

Yet, controlling the runtime behavior of reflection introduces several challenges such as computational overhead [14], the possibility of meta-recursion [6, 7] and an unclean separation of concerns between base and meta-level [2]. These problems arise mainly when implicit reflection (*i.e.,* reflection that is activated *implicitly* by the interpreter on pre-defined execution events [14]) alters the semantics of a running process in ways that lead to excess overhead or inconsistencies.

Although implicit reflection involves complicated modeling concerns, it is useful to consider it – in its most simple form – as operating similarly to an *Event-Condition-Action* model [8, 30]. In a class-based OO language the ECA model would be depicted as shown in Figure 1. The *Event* (left part of Figure 1) in the case of implicit reflection is a semantic action of the underlying interpreter (*e.g.,* read/write slot, message send, method execution etc.) while both *Condition* and *Action* can be considered as custom code snippets (such as block closures) defined by the developer. Registration of such events can take the following form (code presented in Smalltalk syntax):

Script 1: Implicit Reflection Example

```
MsgSend
  when: [ countingFlag = true ]
  do: [ messageCounter := messageCounter + 1 ]
```

In essence, the interpreter triggers an Event (such as a message send) if a predefined condition is met for that event (*e.g.,* a counting flag is set) and then a reflective action is implicitly evaluated (*e.g.,* a message counter is incremented).

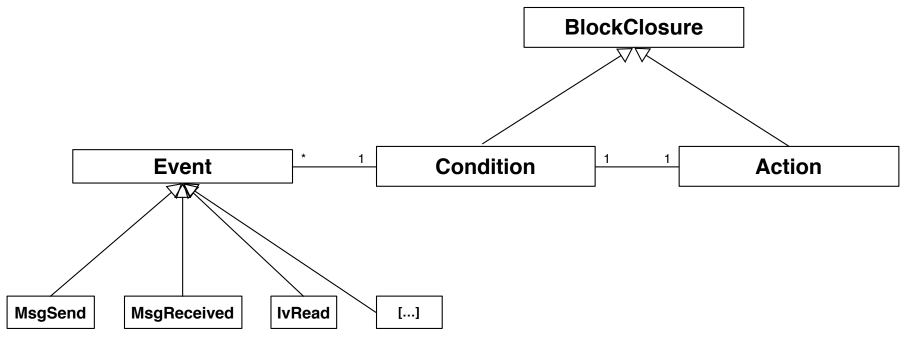

Figure 1: Implicit Reflection as an Event-Condition-Action model

Starting from this general (albeit naive) model for implicit reflection this paper presents five dimensions of meta-level control from related literature, namely: temporal and spatial control, placement control, level control and identity control. We then discuss how these dimensions interact with language semantics in class-based OO languages in terms of: scoping, inheritance and first-class entities. We argue that the reification of the descriptive notion of *reflectogram* [30] can unify the control of meta-level execution in all these five dimensions while expressing properly the underlying language semantics.

The idea of reflectogram was proposed by Tanter et *al.* [30] as a visual depiction of the control flow between the base and the meta-level. Our work proposes to concretize this depiction as an explicit programming language entity. We present an extended model for the reification of the reflectogram based on additional analysis described in this paper. We validate our approach through a new prototype implementation. This revised implementation uses byte-code instrumentation instead of a custom VM, making it compatible with the production Pharo VM and its JIT. Finally, we detail a case study on unanticipated tracing showing that all five dimensions are needed in practical applications and a unified abstraction (such as the reflectogram) is warranted.

The contributions of this work are the following:

- The presentation of different dimensions of meta-control that have been previously treated separately in literature.
- A model for the reification of the reflectogram.
- An implementation of our proposition and its validation.

This paper is an extended version of our previous work [19] that provides the following additional contributions:

- An analysis of the interaction between meta-control dimensions and language semantics in class-based OO languages in terms of: scoping, inheritance and first-class entities.
- Several extensions to the initial model that are based on our additional analysis, including the introduction of the class `ReflectoPoint` which now seperately represents a single point (meta-jump) of the reflectogram.
- A second implementation of our proposition that is based on byte-code instrumentation, on top of Reflectivity[1] for Pharo[2] and is now compatible with a production VM that integrates a JIT. This allowed us to further discuss implementation options comparing to our previous results on top of a JIT-less dedicated VM.

The rest of this paper is organized as follows. Section 2 presents the different dimensions of meta-level control. Section 3 presents our analysis on meta-control

[1]http://scg.unibe.ch/research/reflectivity
[2]http://pharo.org/

and language semantics in class-based OO systems. Section 4 presents our extended model for the reification of the reflectogram. Section 5 illustrates a case study on unanticipated tracing using our approach. Section 6 details the new prototype implementation of our approach in Pharo and compares different implementation options. Section 7 compares related work. Finally, Section 8 concludes the paper and discusses future perspectives.

2. Dimensions of Meta-Control

2.1. Temporal Control

We refer to the *temporal control* of implicit reflection as the ability of runtime installation, activation, de-activation and removal of reflective facilities. Temporal control allows a programmer to define *when* an event will actually be reified by controlling the time of its activation. In essence, setting up meta-actions for events such as the one we described on Script 1 can either be done statically (at compile or load time) or dynamically at run time. In this latter case, trivial conditions like the countingFlag of Script1 are redundant since meta-actions can be installed, enabled, disabled or removed during execution:

Script 2:Temporal Control Example

```
MsgSend do: [ messageCounter := messageCounter + 1 ].
... "code whose messages will be counted"
MsgSend disable
```

Temporal control of reflective facilities at runtime can support unanticipated behavioral reflection as was first illustrated by Redmond et *al.* [22] for the Iguana/J framework [21]. Röthlisberger et *al.* [25] further optimized this approach by supporting unanticipated partial behavioral reflection in Geppetto. Examples include the temporal control of profiling facilities at runtime to facilitate memoization and caching. Röthlisberger et *al.* give such an example for web applications in [25].

2.2. Spatial Control

Spatial control was introduced (as such) by Tanter et *al.* [30] to support a partial reflection scheme in Reflex. This is in spite of the fact that analogous mechanism predate Reflex (such as the CLOS MOP [11]), but which do not distinguish between the different dimensions of their facilities. Spatial control allows a programmer to narrow the scope of implicit reflection to specific entities (objects, classes, methods etc.) and operations thus optimizing performance. In a model supporting spatial control our example from Script 1 would be written as follows:

Script 3: Spatial Control

```
SomeClass
   on: MsgSend
   when: [ countingFlag = true ]
   do: [ messageCounter := messageCounter + 1 ]
```

The difference here on Script 3 (line 1) with our initial example is that a specific class is targeted to be interceded rather than the whole system. Implementations of spatial control — such as the one of Reflex [30] — provide even finer control over what is reified. This is accomplished by targeting single operations on a sub-method level. Other approaches restrict reifications to specific objects over particular executions as in the case of Bifrost [23]. With spatial control unnecessary jumps to the meta-level (*e.g.,* for classes other than SomeClass) can be avoided resulting in an execution speed-up of reflective code.

2.3. Placement Control

On the other hand, *placement control* allows a programmer to define the relative timing of an action in relation to its event as exemplified by Tanter et *al.* [30], but also in works related to method slots [32] and wrappers [3]. For example, user-defined actions can be triggered before or after an event or even totally replace the default semantic action.

Script 4: Placement Control

```
SomeClass
   on: MsgSend
   when: [ countingFlag = true ]
   before: [ Transcript show: 'A message will be sent from SomeClass' ]
   after: [ messageCounter := messageCounter + 1 ]
```

This is shown on lines 4 and 5 of Script 4 where two different meta-actions are registered to be triggered for message sends of SomeClass. The first on line 4 is a logging meta-action registered to be triggered *before* the actual message send at the base-level code of SomeClass. While the second action (line 5) is our counter increment example registered to be performed *after* the event of the message send.

Spatial, temporal and placement control can be used in a variety of contexts where partial reflection is applicable. The most prominent examples can be found in implementations of AOP frameworks [29].

2.4. Level Control

Level control refers to the ability of assigning different reflective behavior to different meta-levels of a reflective tower [31]. Conceptually in OO languages we can say that we operate in a new *"higher"* meta-level whenever a new reflective action is triggered from within the meta-level itself. This process can continue indefinitely if reflective actions are not carefully coded. In this case the problem of infinite meta-recursion occurs [6].

A simple case of infinite meta-recursion illustrating the problem is given on Script 5. On lines 1 to 3, we register (through the message #on:do:) a callback for the MessageReceived event (line 2) of the instance anObject (line1). Essentially, we want the block closure on line 3 to be triggered every time the instance anObject receives a message. Alas, on line 3 in order to increment a message counter (message #incrementMessageCounter) we send another message to the instance anObject from within the meta-level. This new message send will re-trigger the MessageReceived event, re-triggering the callback on line 3, and resulting in an infinite recursion.

Script 5: The meta-recursion problem

```
1 anObject
2    on: MessageReceived
3    do: [ anObject incrementMessageCounter ]
```

Denker et *al.* [7] first proposed a level control mechanism to solve the meta-recursion problem in OO languages through the reification of the metaContext, which represents the level in which a meta-jump occurs. The metaContext is an implicit entity of the meta-level, in the sense that the developer does not invoke it explicitly but rather executes code or binds meta-objects to specific meta-levels (as shown on Script 6). As described in [7] in order to control the meta-level and avoid meta-recursion, Denker et *al.* provide a way to run a block-closure one meta-level higher than the code outside the block as follows:

Script 6: Code execution with a metaContext [7]

```
[ ... code executing on meta−1 ... ] valueWithMetaContext
...
[
   [ ... code executing on meta−2 ... ] valueWithMetaContext

] valueWithMetaContext
```

More recently and in another context (that of AOP) the idea of *execution levels* has been proposed [27, 28]. These execution levels provide a concrete solution to the problem of *aspect loops* (the equivalence of meta-recursion in AOP) for languages such as AspectScript [20] and AspectJ [12].

Besides being a solution to the meta-recursion problem, level control can prove useful in other contexts of meta-circularity. Examples include the profiling of meta-level execution itself through reflection.

2.5. Identity Control

When base-level objects are responsible for their own reflective operations it is hard to allow finer control over reflective behavior. Identity control is the factoring of reflective operations out of the targeted base-level objects. This decomposition was investigated by Bracha and Ungar through Mirrors [2] but has been studied in different contexts as well [9, 4]. AmbientTalk [17] was the first mirror-based implementation specifically targeting implicit reflection.

With identity control the meta-object (mirror-based or otherwise) assumes the identity of the receiver of the reflective method, bypassing the object's control altogether. From this perspective the problem is also closely related to the ideas of both subject and context-oriented programming [34, 33, 35]. Indeed these two paradigms deal with variations of object behavior that conceptually do not depend on one specific object, but rather on the execution context and its state.

Identity control can prove useful in situations such as the one depicted in Figure 2. In Figure 2 anObjectInpector wants to inspect the slots of a base-level object. This object (aPersistentObject) supports persistency (on a file or on a database) through reflective intercession. This means that the semantics for accessing instance variables of a persistent object have changed through reflection to synchronize the state of the object with external data storage. Let us now assume that this was achieved by instructing the compiler to transform each read and write access to instance variables into meta-level calls. For example, in this case each read access of instance variables in the class PersistentObject will be redirected to the meta-level method #instVarAt which has been overridden from Object to provide the additional functionality.

Although this change in semantics for aPersistentObject is desirable, it does not make sense in the case of anObjectInspector. The object anObjectInspector (as do most programming tools) wants direct access to the slots of aPersistentObject without triggering the back-end (*i.e.,* database) logic. Bracha and Ungar suggest that for such cases reflective facilities that are not part of the language kernel should be used. In this example a separate read access method from the one depicted in Object»#instVarAt: — and which the ObjectInspector»#instVarAt: invokes — can be used.

This is the case of the #objInstVarAt: method of ObjectInspector which has read access through direct virtual-machine support to object slots bypassing the targeted object's control. What has essentially changed here between Object»#instVarAt: and ObjectInspector»#objInstVarAt: — which perform the same operation — is the *identity* of the receiver of the reflective action. In the first case the receiver is aPersistentObject while in the latter it is anObjectInspector.

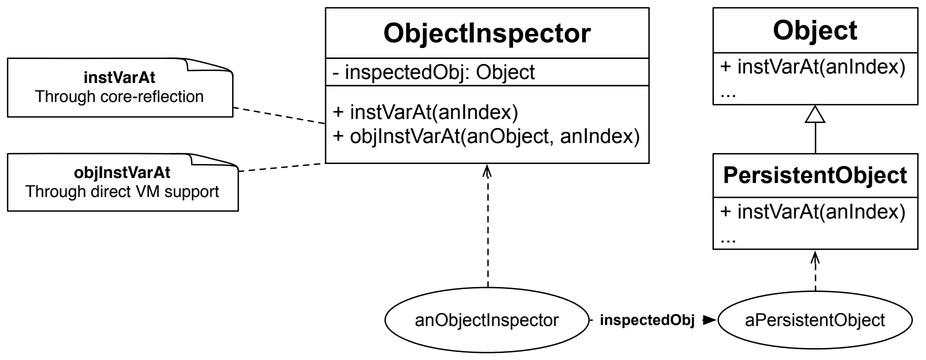

Figure 2: Inspecting an intercessed object

In summary, identity control besides solving problems as the one we described above promotes a stricter separation of concerns between base-level and meta-level functionality.

3. Meta-control and Language Semantics

Using meta-level facilities as the ones we described in Section 2 presents certain challenges in class-based OO languages. In this section we report and categorize these issues, using mainly an instance and a class entity as our running examples. Nevertheless, similar issues apply to other OO entities such as environments, packages, methods, processes and contexts.

3.1. Meta-control and Scoping Semantics

The ECA model for implicit reflection we saw in Section 1 describes events that are implicitly triggered at specific program execution points. The definition of these execution events though are highly dependent on the concrete implementation of a reflective framework. While the developer may expect an event of the form "anEntity on: <Event> do:..." to always behave the same way, we will shortly see that an <entity[3], event> pair does not unambiguously define a reflective operation.

Our first example on Script 7a below involves a terminal instance and two events (on lines 1 and 3 respectively):

Script 7a: Scoping: Reducing vs. Specializing

```
1 anInstance on: MsgSend do: [...]
2
3 anInstance on: PassedAsArg do: [...]
```

Now, on line 1 of Script 7a above, the event MsgSend (a method invocation) which by itself has a global scope (can happen anywhere) is reduced to the scope of a specific object. Script 3 also shows an example of such a scoping.

But then on line 3 the event PassedAsArg (passing a reference of an entity in a dynamic method invocation) which (as with MsgSend) has by itself a global scope (can happen anywhere) may have different interpretations. On the one hand as before (line 1) it may be interpreted as *reducing* the scope of the event, to the object's scope. On the other hand though it may be interpreted as *specializing* the event (*i.e.,* the event remains global but is only triggered if this specific object is passed as an argument). Indeed passing even a specific object as an argument (in a dynamic environment) can happen anywhere.

So far we can say that depending on the event, meta-control can either reduce the global scope of an event or specialize it with a condition (while it remains global). The situation becomes a little more complicated on Script 7b. Here, we have the same event as line 3 of Script 7a but the entity is now the class of anInstance. In this case, depending on the implementation we can either:

Script 7b: Scoping: Reducing *vs.* Specializing

```
1 anInstance class on: PassedAsArg do: [...]
```

[3]instance, class, package etc.

1. Treat the class as a scope and more specifically, as a set of lexical scopes (methods) from which we will intercess all PassedAsArg events (regardless of which object is passed), similarly to the event of line 1 (ie reduce the scope).
2. Treat the class as a set of runtime entities and trigger the PassedAsArg event whenever an object of the specific class is passed as an argument, as in the event of line 3 (ie specialize).
3. Finally, if the underlying language supports first-class classes, there is a third option: Treat the class as an object itself and trigger the PassedAsArg event whenever the class itself is passed as an argument.

3.2. Meta-control and First-class entities

This last case (first-class entities) is the intended mode of operation of our second example below (Script 8). In this example, starting at line 1, the MsgReceive event is specialized by the first-class instance of (anInstance class) with the condition that the dynamic method invoked has the signature #basicNew (line 3). Essentially, this snippet simulates the ObjectCreation event (signaled whenever an instance of a specific class is created), showing that this third choice (of treating classes as first-class in implicit reflection) is not a marginal but a very useful case.

Script 8: Simulating ObjectCreation

```
1  anInstance class
2    on: MsgReceive
3    when: [:selector |
4              (selector = #basicNew) | (selector = #basicNew:)]
5    do: [...]
```

3.3. Meta-control and Inheritance

Finally, we discuss inheritance. Consider a class hierarchy (as the one depicted in Figure 3 originating from Object with a subclass A then a subclass of A named B and finally, a subclass of B named C. Assuming that MsgReceive here treats its target (say the class B) as a set of runtime entities, what does a reflective MsgReceive event (as the one we saw in the above snippet) mean when applied to class B for instances of C and for the methods of A ?

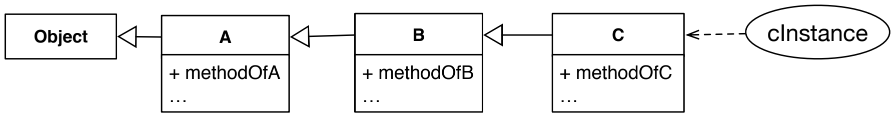

Figure 3: A simple hierarchy with class B as a reflection target

Here we have the following (canonical) outcome:

- Concerning messages received by instances of B that invoke methods inherited from A the canonical response is to intercept them.
- Concerning instances of C the canonical approach is to intercept only those messages that invoke methods of B and A (but not methods of C).

However, this is not always the desired outcome. Consider, for example, that the canonical approach here will also intercess all methods of Object (the root of the hierarchy) for the instances affected. This is a problem as Object includes in most systems critical system methods that should not be intercepted. Propagating reflective behavior automatically inside the hierarchy chain like that may not be the desired outcome, especially for long hierarchies where the developer is subclassing without actual knowledge of the inherited functionality. Similar issues arise in class-based OO languages for other events such as IVRead (instance variable read) and IVWrite (instance variable write).

3.4. Summary

In summary, given the above ambiguities in terms of scope, first-class entities and inheritance, we believe that models of implicit reflection should make the distinctions mentioned in this section between different cases explicit. Alternatively reflective frameworks should allow developers to decide for each case programmatically.

4. Reifying the Reflectogram

The notion of reflectogram was introduced by Tanter et *al.* [30] as a conceptual illustration to describe meta-level behavior to human readers:

[...] A reflectogram illustrates the control flow between the base-level and the meta-level during execution.

For example, in the left part of Figure 4 we see a diagram from Tanter et *al.* describing spatial control and partial reflection, while in the bottom part of Figure 4 we see a depiction from the same paper of temporal control. Other researchers have used similar control flow illustrations to describe different dimensions of meta-level behavior as the diagram we reproduce from [7] (right part of Figure 4) describing level control.

Given the versatility of the reflectogram for describing meta-level behavior, the rest of section describes our proposition of reifying it as an explicit meta-object. This meta-object is passed as an argument at runtime to conditions and implicit actions invoked by the underlying execution environment. Our goal is to unify the control of meta-level execution under a single abstraction for end-users.

4.1. Reifying the Reflectogram

Our proposal (shown in Figure 5) in its more general form extends the *Event-Condition-Action* model of implicit reflection (depicted in Figure 1) by establishing a one-to-many relation of both the condition and the action with the

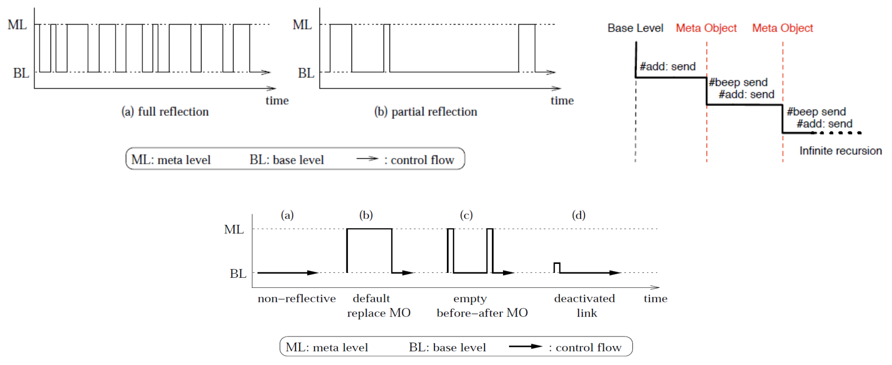

Figure 4: Diagram of the reflectogram in literature: Tanter et *al.* (left and bottom) [30] and Denker et *al.* [7] (right)

reified reflectogram. More specifically, this relationship is between the reflectopoint (reifying a single point of the reflectogram) and the meta-conditions/actions. The relationship is one-to-many in the sense that a single action or condition can be registered for multiple objects. Upon each invocation, the reflectogram corresponding to the particular object that triggered the event creates a new reflectopoint that will be passed as an argument to the meta-level.

Besides holding a reference to that targetObject, the reflectogram should provide meta-information through each reflectopoint for the currently triggered event which — depending on the implementation — can be used to parametrize conditions and actions at runtime.

Compared with our initial model [19] the introduction of the reflectopoint makes explicit the fact that there is a new meta-object generated for each meta-jump. This meta-object can now control the outcome of individual reflective operations. This allows us to make a clear distinction between:

- Reflectopoints, representing single-points (jumps) on the reflectogram
- and the reified reflectogram itself, which through event registration controls the overall evolution in time of a target object.

Furthermore, in order to address the issues identified in Section 3 we introduce one level of indirection between the ReflectoGram and its targeted object through the extended class Target described in detail in Section 4.3.

Event Registration. The reflectogram controls the spatial dimension of implicit reflection at runtime through the methods #on:when:do: and #on:for:do:when:. The latter method is either a static or a class-side method depending on whether classes in the target language are first class or not. Registering an event for a specific object can be modeled as follows:

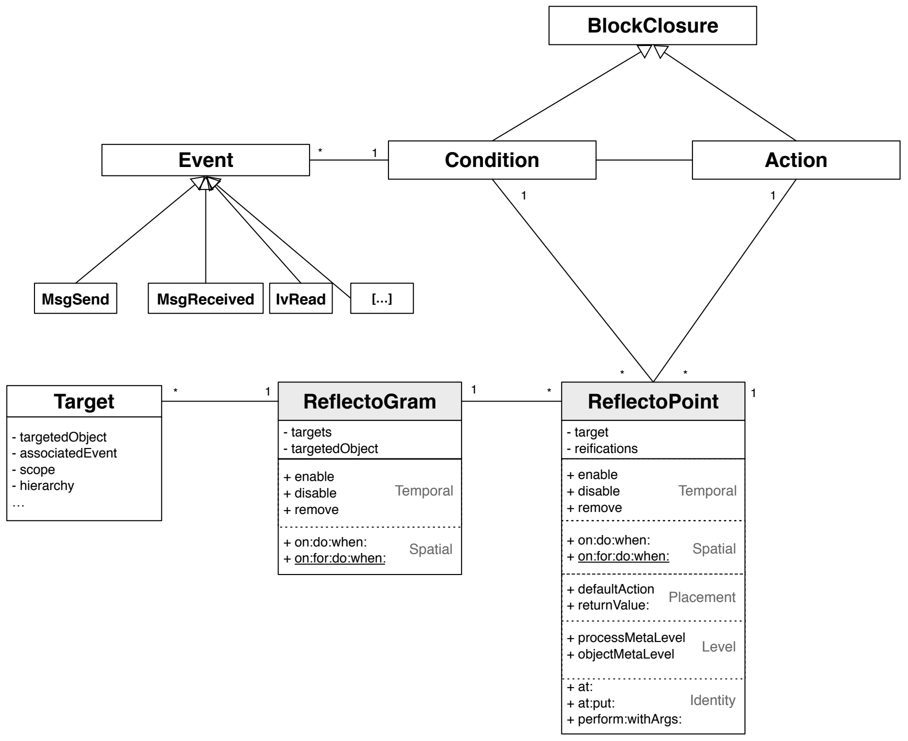

Figure 5: Our extended proposal: Reifying the Reflectogram

Script 9: Registering events with the reflectogram

```
Reflectogram
   on: anObject
   for: MessageReceived
   when: [:reflectopoint | "condition" ]
   do: [:reflectopoint | "action" ]
```

As seen on lines 4 and 5 of Script 9, conditions and actions in our model receive an argument (reflectopoint) which describes and controls the "shape" of the reflectogram for each meta-level jump of the associated base-level object. The class-side method #on:for:do:when: is used for the initial registration of an event. Its instance-side counterparts #on:when:do: provides the same functionality in order to register more events at runtime from within the meta-level.

It is worth noting here that while we are assuming a single meta-level for our system, both conditions and actions depicted in Figure 5 can be considered as first-level implicit meta-objects. That is, objects that are invoked implicitly during specific events by the underlying execution environment [14]. From this perspective both the reflectogram and the reflectopoints can be seen as second-level explicit meta-objects, in the sense that their domain is the control of the first implicit meta-level of conditions and actions.

4.2. The Reflectogram API

The API of our model is organized into five distinct protocols corresponding to the five dimensions for meta-level control discussed in Section 2:

Temporal Protocol. Methods #enable, #disable and #remove as their name suggests control the actual triggering of events. Implementors can choose to provide static counterparts for convenience (such as #enableFor:, #disableFor: etc.).

Spatial Protocol. Methods #on:when:do:, #on:for:do:when: control spatial selection by registering events for specific objects as it has been described above.

Placement Protocol. Methods #defaultAction and #returnValue: control the placement of *meta-actions.* The reflectogram can invoke the *default* base-level action from within the meta-level thus implicitly defining which meta-level statements will be executed *before* and which statements *after* the actual event. Regardless of whether the default action has been triggered from within the meta-level the value that will be returned to the base-level can be explicitly set, thus facilitating total replacement of base-level semantics.

Level Protocol. Methods #processMetaLevel and #objectMetaLevel return the height of the currently executing meta-level action or condition as in the meta-level tower model. The #processMetaLevel returns the process-wide meta-level height, while the #objectMetaLevel returns the number of meta-levels that have been triggered due to events of the reflectogram's target object.

Identity Protocol. Finally, methods #at:, #at:put: and #perform:withArgs: provide read, write (for slots) and execution reflective facilities (for message sending) for the target object. These methods are implemented separately from core reflection and their corresponding message sends are received by the reflectopoint rather than the target object. This way the identity of the receiver of reflective methods is controlled as was described in Section 2.5.

A usage example of the reflectogram is depicted on Script 10 where we solve the meta-recursion problem that was described in Section 2.4 (Script 5) by explicitly controlling the meta-level execution flow:

Script 10: Solving the meta-recursion problem with the reflectogram

```
Reflectogram
   on: anObject
   for: MessageReceived
   when: [:reflectopoint | countingFlag = true ]
   do: [:reflectopoint |
           reflectopoint disable.
           anObject incrementMessageCounter.
           reflectopoint returnValue:
                  reflectogram defaultAction.
           reflectopoint enable.
   ]
```

Lines 1 to 3 of Script 10 register the MessageReceived event for the instance anObject. On line 4 — as before — a trivial condition is registered checking a message-counting flag. Then on lines 5 to 11 a meta-action is registered for the MessageReceived event. On line 6 the reflectopoint is explicitly disabled thus temporarily allowing message sends to be received by anObject without interception. On line 7 the message #incrementCounter is send to anObject without resulting in an infinite recursion since the reflectopoint has been disabled. Then on lines 8 to 9 the value that will be returned to the base-level is set to the default semantic action for MessageReceived events. This default action corresponds to the evaluation of whichever message send (received by anObject) was intercepted and triggered the meta-jump. Finally, on line 10 the reflectopoint is re-enabled before returning control to the base-level, as to be able to intercept further message sends to anObject.

4.3. Handling Scoping, Reifications and Inheritance

In Figure 5 although each ReflectoGram is still associated with a single target object (as with our initial model) it has now multiple Targets (for the same object) which differentiate between different events, scopes, hierarchy policies etc. This class Target introduces one level of indirection between the ReflectoGram and the targetedObject in order to address the issues identified in Section 3 concerning scope, inheritance and first-class entities.

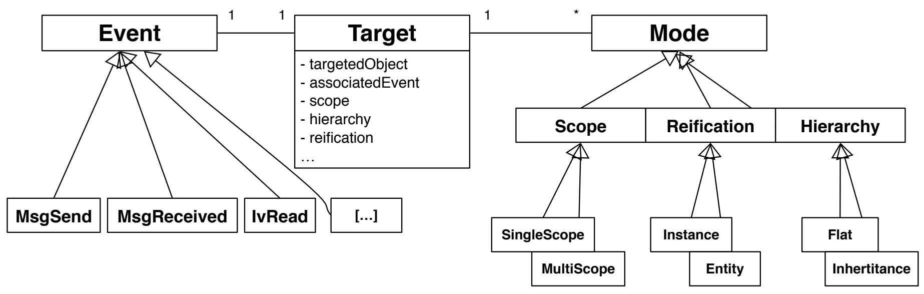

Figure 6: Different modes for targets of the reflectogram

In Figure 6, we see that each target apart from each association with exactly one event, stores multiple *Mode objects* (scope, hierarchy, reification slots) that disambiguate the exact semantics of an <event,target> pair according to the following rules:

Scope (issue described in Section 3.1)

- **SingleScope** Treat the target as a single logical scope (even if this is a composite scope). For example, in Script 7b, the class of line 1 will be treated as a *single logical scope* comprised of a set of methods.
- **MultiScope** Treat the target as a set of runtime entities. For example, in Script 7b, the class of line 1 will be treated as a set of its instances, and trigger the PassedAsArg event whenever (from multiple scopes) an object of the specific class is passed as an argument.
- **By Default** Treat the target with default scoping mode defined in the event itself. For example, a PassedAsArg event can always be a multi-scope event (concerning only sets of runtime objects and not scopes). This option partially solves the problem by making the convention explicit.

Reification (issue described in Section 3.2)

- **Instance** Treat the target as an instance (*i.e.,* if it is first-class) and not as a set of either scopes or other instances. For example, in Script 8, the class is treated as a terminal instance to simulate the ObjectCreation event.
- **Entity** Treat the target as the semantical entity it represents (*i.e.,* a class as a set of objects or methods, a package as a set of classes etc.), following the scoping rules (above).
- **By Default** We believe that treating all reifications as entities is the most logical default behavior. Nevertheless, as we show in Section 3.2 it is useful to allow the programmer to override this.

Hierarchy (issue described in Section 3.3)

- **Flat** Ignore the class hierarchy and operate only on the set of either methods or objects belonging to the target.
- **Inheritance** Propagate inheritance rules to the hierarchy as is described in Section 3.3.
- **By Default** We believe that by default all targets should be treated as Flat to avoid surprises while propagating reflective behavior, but nevertheless allow end-users to override it if necessary.

From the point of view of the end-user the above model can be integrated in an extended event registration API as follows:

Script 11: Extended Event Registration: Handling different modes programmatically

```
1 Reflectogram
2   on: aTarget
3   as: Entity —> Flat
4   for:  PassedAsArg "class inheriting from MultiScopedEvent"
5   limitedTo: [:method | "conditon limiting installation in a set of methods"|]
6   when: [:reflectopoint | "reflective condition" ]
7   do: [:reflectopoint | "reflective action" ]
```

Here, the programmer can optionally set the policies for reification and hierarchy (line 3). Furthermore, scoping (line 4) is made explicit by organizing events into scoping hierarchies (*i.e.,* the PassedAsArg inherits from MultiScopedEvent). Finally, the limitedTo: directive on line 5 can programmatically restrict the propagation of the reflective behavior (in a set of methods) even in the case that an inheritance mode is set as the hierarchy policy.

5. The Reflectogram in Action

This section presents the implementation of a non-trivial tracing framework where the code that will be traced is not known a priori (*i.e.,* is unanticipated) but is being instrumented on the fly at runtime. Message-based profiling [1], for example, uses such a tracing approach to approximate execution time of selected methods. Through this example, we aim to show that all five dimensions of control co-occur in practical applications and a unified abstraction (such as the reflectogram) is warranted. In the code snippets below the extended revised event registration described in Section 4.3 is used, while the event MsgSend inherits here from SingleScopedEvent.

Figure 7 shows the core classes of our tracing framework, which include:

CallGraph. The entry point of the output call graph of our tracing process.

CallGraphNode. Individual nodes of the output call graph holding the actual meta-information that has been traced. For our framework, this meta-information includes: *the receiver of a message send, its class, the selector and the arguments passed along with the message call.*

ExecutionTrace. Users subclass ExecutionTrace adding the entry point symbol of the code to be traced by invoking the inherited #run: aSymbol method (where aSymbol corresponds to a method name). Also inherited are the corresponding output call graph and the process (*i.e.,* green thread) where the tracing of a targeted method will take place.

CallTrace. Finally, CallTrace implements the condition and send callbacks (Script 12) which are bound to traced objects at runtime. These callbacks then delegate meta-level control to methods #inTracingScope:, #addGraphNode:, #executeNode: and #return:, respectively (Script 13).

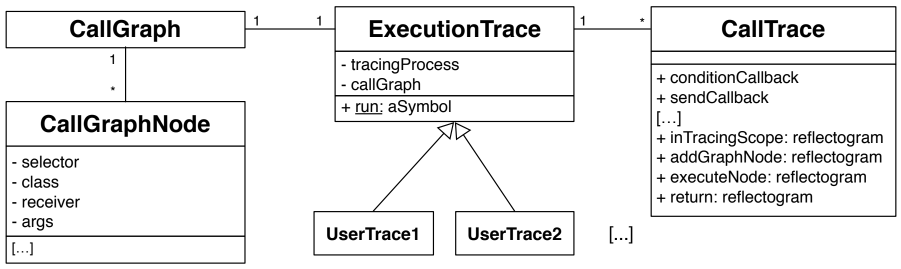

Figure 7: Core classes of our tracing framework

As seen on lines 2,7,8 and 9 of Script 12 since the reflectogram and its points are reified as first class entities they can be passed as arguments beyond the scope of conditions or meta-actions. On line 2, a reflectopoint is passed to the #inTracingScope: method to determine if the meta-event that triggered the conditionCallback happened inside our tracing process or not. On line 7, it is passed to #addGraphNode: to gather the meta-information needed to update the call graph. Then, on line 8 it is passed to #executeNode: to perform the default base-level action and gather its return value. Finally, on line 9 it is passed to the #return: method which resets both the call graph and the reflectogram appropriately for further processing.

Script 12: Tracing Callbacks

```
1  conditionCallback
2     ^ [ :reflectopoint | self inTracingScope: reflectopoint ]
3
4  sendCallback
5     ^ [ :reflectopoint |
6              reflectopoint disable.
7              (self addGraphNode: reflectopoint)
8                  returnValue: (self executeNode: reflectopoint).
9              self return: reflectopoint ]
```

On Script 13 we see these delegate methods in more detail:

Script 13: Meta-control methods using the Reflectogram

```
1   inTracingScope: reflectopoint
2     ^ reflectopoint reifications process =
3      tracingProcess & (reflectopoint processMetaLevel = 1)
4
5   addGraphNode: reflectopoint
6     ^ callGraph
7           addSelector: reflectopoint reifications message selector
8           forClass: (Reflectogram
9                       object: reflectopoint reifications receiver
10                      perform: #class
11                      withArguments: #())
12          rcvr: reflectopoint reifications receiver
13          args: reflectopoint reifications message arguments
14
15  executeNode: reflectopoint
16    newCallTrace := self class
17          newWithGraph: callGraph
18          forProcess: tracingProcess.
19    ReflectoGram
20          on: reflectopoint reifications receiver
21          as: Entity —> Flat
22          for: MsgSend
23          limitedTo: [:method | true ]
24          when: [:rpoint | newCallTrace conditionCallback value: rpoint ]
25          do: [:rpoint | newCallTrace sendCallback value: rpoint ].
26    ^ reflectopoint
27          returnValue: reflectopoint defaultAction
28
29  return: reflectopoint
30     callGraph return.
31     reflectopoint enable.
```

Method #inTracingScope: (lines 1 to 3) the *level protocol* of the reflectopoint is used (line 3) in order to determine whether we are intercepting a method call that originated from the base-level of our tracing process (processMetaLevel = 1). If not #inTracingProcess: will return false and the corresponding meta-action (lines 5 through 9 on Script 12) will not be invoked.

Method #addGraphNode: (lines 5 through 13) the reification slot of the reflectopoint is used in order to gather meta-information about the intercepted call and update the call graph. The reification slot stores event-specific information for each meta-jump. This information can include: the sender, receiver, arguments and context of message sends, name and values of variables accessed, the active process, the targeted object, the type of the event etc. Method calls are intercepted every time a message is sent to a new receiver (from within a traced object). On lines 8 through 11 the *identity protocol* is used in order to extract the class of this new receiver and avoid the meta-recursion problem in case this receiver was previously being traced.

Method #executeNode: (lines 15 to 27) a new call trace is being created and is being assigned to the new receiver at runtime via the *spatial protocol* (lines

19 to 25) using the extended event registration API. Then on lines 26 to 27 the *placement protocol* is being used to perform the default base-level action and gather its return value. Since the base-level action is a method call to a newly-traced object it will re-trigger the meta-level for all new method calls from within its scope before returning.

Method #return: (lines 29 to 31) is the equivalent of an after-action. Here we update the call graph (to point to the node that we have previously added) and re-enable the reflectopoint (line 31) for our traced object through the *temporal protocol*. The reflectopoint had been previously disabled for convenience (in order to avoid unnecessary meta-jumps) on the beginning of the meta-action callback (line 7, Script 12).

6. Implementation

Previously, the prototype implementation of our model for the reflectogram was part of a dedicated virtual machine targeting the Pharo platform: the metaStackVM[4] [18]. Essentially, through VM support, instrumentation checks can be performed on the objects themselves while they are being interpreted by the underlying execution environment. We implemented the metaStackVM by extending the standard Stack VM[5] [16] of Pharo, in Slang [10]. Slang uses a subset of the Smalltalk syntax with procedural semantics that can be easily translated to C.

Currently, we report on experimentations made with a new implementation of our proposition using byte-code instrumentation, on top of Reflectivity[6]. This fact made our current implementation compatible with a production just-in-time VM, and allowed us to further discuss implementation options comparing to our previous results. Reflectivity itself is based on the Gepetto model [24, 7] which uses lower-level abstractions (such as links and hooksets) apart from that of meta-objects in order to provide a stricter separation of concerns between handling of events (hookset responsibility) and meta-level delegation (link responsibility). More specifically, byte-code instrumentation in Reflectivity is performed at runtime (as in [24]) using extension modules of the Opal compiler[7] allowing us to define:

- Sets of related base-level events that are described by reified objects (such as abstract syntax nodes).
- Meta-objects (any object can play this role) to which control will be delegated at runtime.

[4]http://ss3.gemstone.com/ss/mSVM.html
[5]https://ci.inria.fr/pharo/view/VM/job/PharoSVM/
[6]http://scg.unibe.ch/research/reflectivity
[7]http://scg.unibe.ch/research/OpalCompiler

- A link between the events described by the AST nodes and the meta-object. Links can dynamically control the activation/deactivation of meta-level behavior.

Our own model does not cover (but uses) these concerns to offer higher-level abstractions that are more suitable for end-users. The rationale for our current choice to base our solution on top of a byte-code instrumentation framework is twofold. On the one hand, a customized VM (as in our previous implementation) is not a viable solution in the long run, unless the extensions become part of the vendor's VM for the platform. On the other hand, our previous experiment depended on a JIT-less VM which puts a performance penalty when compared to the state of the art. Both implementations of the reflectogram are available at http://ss3.gemstone.com/ss/mSVM.html together with the benchmarks presented in this section.

We evaluated both solutions though a micro-benchmark to measure the overhead introduced to normal execution with and without reifying the reflectogram. The benchmark is based on Tanter et *al.* [30] and measures the slowdown introduced for 10^6 messages sent to a test object when a) no instrumentation is present b) instrumentation is loaded but is disabled c) instrumentation is enabled d) instrumentation is enabled and the reflectogram is being reified.

For each platform, we used the most direct messaging event depending on the implementation. For example the metaStackVM uses the MessageReceive event send for the test object itself (since the VM has access to the runtime instances). Our current implementation uses the MessageSend event, which is a messaging event from within a specific scope (*i.e.,* for a method that is being instrumented).

We present the results relative to normal execution for each platform individually measuring the slowdown introduced.

We report average execution times $\mu = \frac{1}{N}\sum_{i=1}^{N} t_i$ in milliseconds (ms) over 100 runs of the bechmarks (each run comprising 10^6 message sends), using the standard deviation $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(t_i - \mu)^2}$ as a measure of uncertainty/error for the measurements.

The average slowdown[8] is calculated as a derived quantity z of two directly measured variables x, y, where $z = x/y$, whence the standard deviation of z can be calculated as a propagation error using the formula: $\frac{\Delta z}{z} = \sqrt{\left(\frac{\Delta x}{x}\right)^2 + \left(\frac{\Delta y}{y}\right)^2}$. Where z is the the derived quantity (slowdown) and x, y the directly measured averages of the benchmarks.

In order to avoid skewing the results by the warmup time of the JIT we followed a similar procedure as the one described by J. Ressia for Bifrost [23] namely making sure that:

- Benchmarked methods are created in advance.

[8]reported as *x times* the speed of normal execution

10^6 msg-sends over 100 runs – SLOWDOWN (Bytecode-JIT)				
Case	Time μ (ms)	Time σ (ms)	Slowdown μ	Slowdown σ
No instrumentation	9.00 ms	0.00 ms	**1.00**	0.00
Disabled instrumentation	9.00 ms	0.00 ms	**1.00**	0.00
Before-hook	11.02 ms	0.14 ms	**1.22**	0.01
Instead-hook	77.91 ms	0.97 ms	**8.66**	0.11
Reflectogram Reification	188.87 ms	6.73 ms	**20.98**	0.75
Reflectogram / Instead	–	–	**2.42**	0.09

Table 1: Instrumentation Benchmark with Bytecode Instrumentation in the presence of a JIT

10^6 msg-sends over 100 runs – SLOWDOWN (mSVM-no-JIT)				
Case	Time μ (ms)	Time σ (ms)	Slowdown μ	Slowdown σ
No instrumentation	48.10 ms	0.52 ms	**1.00**	0.00
Disabled instrumentation	47.52 ms	0.90 ms	**0.99**	0.02
Enabled instrumentation	414.08 ms	22.07 ms	**8.61**	0.48
Reflectogram Reification	658.08 ms	69.91 ms	**13.68**	1.46
Reflectogram / Instead	–	–	**1.59**	0.19

Table 2: Instrumentation Benchmark with a dedicated VM, inline-caching no-JIT

- Method lookup tables are filled by repeated execution before the measurements.
- Measured methods compute constant time arithmetic operations for which there are no further optimisations provided by the VM.

As we see in Tables 1 and 2 when instrumentation is loaded into the environment but is disabled for the benchmarked object, there is no additional overhead. This is important for practical reasons to avoid slowing down the whole system when instrumenting only a part of it [30]. For example, implicit reflection in both cases introduces an 8.66x (for bytecode instrumentation with Reflectivity) and 8.61x overhead (for our non-JIT customized VM) *but only* for the benchmarked object operating in an instead mode with reifications[9]. In fact for bytecode instrumentation with Reflectivity when only a before directive is used[10] without any reifications for events the slowdown introduced can be as low as 1.22x.

But while adding the reification of the reflectogram to the metaStackVM introduces a 1.59x slowdown compared to implicit reflection without such reification, in the case of bytecode instrumentation the additional slowdown for using the reflectogram is 2.42x. Nevertheless the added benefit of having the reflectogram operating on top of the vendor VM (in the presence of JIT compilation)

[9]meaning that the reflective action fully replaces base-level functionality

[10]meaning that the reflective action is executed before the base-level functionality, but not replacing it

with a faster underlying infrastructure allows for further optimizations [15] that we have not yet explored. Our results point us towards a synergetic approach, where both bytecode instrumentation and the VM are involved.

7. Related Work

In Section 2 we presented five dimensions of meta-control that have been previously treated separately in literature. Table 3 summarizes the facilities of their corresponding implementations and compares them with our implementations of the reflectogram.

	Iguana/J [21, 22]	**Reflex** [30]	**Gepetto** [24]	**Gepetto-Ext** [7]
Temporal	✓	✓(partially)	✓	✓
Spatial	✓	✓	✓	✓
Placement	×	✓	✓	✓
Level	×	×	×	✓
Identity	×	×	×	×

	AmbientTalk [17]	**Bifrost** [23]	**Reflectogram**
Temporal	✓	✓	✓
Spatial	✓	✓	✓
Placement	×	✓	✓
Level	×	×	✓
Identity	✓	×	✓

Table 3: Comparison in terms of Meta-Control Facilities

While Iguana/J [21, 22] was the first to introduce unanticipated changes (temporal control) and spatial control for the Java platform, it was Reflex [30] that introduced partial behavioral reflection (supporting spatial, temporal and placement control). On the other hand, due to implementation constraints Reflex did not allow for dynamic definition of meta-level behavior at runtime as did Iguana/J. This is in spite of the fact that analogous mechanisms predate both Iguana/J and Reflex (such as the CLOS MOP), but do not distinguish between the different dimensions of their facilities. These three types of control (placement, spatial and temporal) were first available during runtime in Gepetto for Smalltalk [24], with later extensions to the Gepetto implementation covering level control [7]. Bifrost added an object-centric model to runtime reflection expanding spatial control to execution runs [23]. AmbientTalk [17] was the first mirror-based implementation specifically targeting implicit reflection and has support for temporal, spatial and identity control.

Our own implementation manages to cover all five dimension of meta-control through the reflectogram reification. It is close to Gepetto [24] but additionally deals with identity control [7].

Aspect-Oriented Programming. As discussed in Section 2, the problems presented in this paper have direct analogies to issues presented in AOP literature. In the context of AOP, the dimensions of spatial, temporal and placement control are embedded in the abstractions of aspects, namely: advices and join points. Moreover, the recent proposal of *execution levels* [27, 28] solves the equivalent problem of meta-recursion by avoiding aspect loops.

Subject and Context-Oriented Programming. Finally, as we discussed in Section 2.5, the problem of identity control is closely related to the ideas of both subject and context-oriented programming [34, 33, 35]. These two paradigms deal with variations of object behavior that conceptually do not depend on one specific object. On the contrary these variations are linked to the general execution context and its state. Our solution, instead of providing an implicit variation of behavior on the target object, gives the programmer an alternative way to explicitly access its reflective behavior. With identity control the meta-object assumes the identity of the receiver of the reflective method, bypassing the object's control altogether. This basic approach can nevertheless be used to build more elaborate abstractions.

Limitations. From a model perspective, our solution presents some limitations compared to the model of Reflex or Gepetto, which are focused on extensibility. These models introduce abstractions (such as links and hooksets) apart from that of meta-objects in order to provide a stricter separation of concerns between event handling (hookset responsibility) and meta-level delegation (link responsibility). Other solutions such as Bifrost provide additional support for compound meta-objects allowing for composition of meta-behavior. Our approach presents a single unifying entity (the reflectogram) for meta-level control aiming at explicit handling of control flow from within the meta-level itself.

From this perspective, the reflectogram is more appropriate as an end-user abstraction rather than an implementor's abstraction since it does not focus on extensibility or composition. On the other hand, the reflectogram is described through the Event-Condition-Action model which all implicit reflection schemes (including Bifrost, Gepetto and Reflex) share and can thus be implemented - as we showed through our current implementation - as an extension on top of them.

8. Conclusion

Our work presents five different dimensions of meta-control for implicit reflection that have been treated separately in literature, namely: temporal and spatial control, placement control, level control and identity control. We discuss how these dimensions interact with language semantics in class-based OO languages in terms of: scoping, inheritance and first-class entities. We have proposed an extended model for the reification of a previously purely descriptive notion – that of the reflectogram [30] – arguing that such reification can unify the control of meta-level execution in all five dimensions, while expressing properly the underlying language semantics. We have also presented an additional prototype implementation of this reification in the Pharo programming environment which uses byte-code instrumentation - and is now compatible with a production VM that integrates a JIT compiler. We have validated our approach through a case study on unanticipated tracing.

In terms of future work, we would like to provide a formal semantic representation of the reflectogram and experiment with a more synergetic approach for implementing it where both bytecode instrumentation and the VM are involved. Finally we would like to provide benchmarks for larger examples and programming tools in a way where the impact of the framework itself is measured (as in our micro-benchmarks) rather than the slowdown of the meta-level application logic.

9. Acknowledgments

The authors would like to thank Prof. Eric Tanter for his valuable feedback - on the previous version of this work - regarding related literature in the context of AOP and for helping us strengthen our presentation. We will also like to thank our anonymous reviewers for their thorough work and constructive suggestions. This work was partially sponsored by the European Smalltalk Users Group: http://esug.org.

[1] Bergel, A., 2011. Counting messages as a proxy for average execution time in Pharo. In: ECOOP 2011. Vol. 6813 of LNCS. Springer, pp. 533–557.

[2] Bracha, G., Ungar, D., 2004. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In: OOPSLA 2004, ACM SIGPLAN Notices. ACM Press, New York, NY, USA, pp. 331–344.
URL http://bracha.org/mirrors.pdf

[3] Brant, J., Foote, B., Johnson, R., Roberts, D., 1998. Wrappers to the rescue. In: ECOOP 1998. Vol. 1445 of LNCS. Springer-Verlag, pp. 396–417.

[4] Cazzola, W., 2003. Remote method invocation as a first-class citizen. In: Distributed Computing 16 (4), 287–306.
URL http://dx.doi.org/10.1007/s00446-003-0094-8

[5] Cazzola, W., Chiba, S., Ledoux, T., 2000. Reflection and meta-level architectures: State of the art and future trends. In: ECOOP 2000. Vol. 1964 of LNCS. Springer-Verlag, pp. 1–15.
URL http://dl.acm.org/citation.cfm?id=646780.705788

[6] Chiba, S., Kiczales, G., Lamping, J., 1996. Avoiding confusion in metacircularity: The Meta-helix. In: Futatsugi, K., Matsuoka, S. (Eds.), Proceedings of ISOTAS '96. Vol. 1049 of LNCS. Springer, pp. 157–172.
URL http://www2.parc.com/csl/groups/sda/publications/papers/Chiba-ISOTAS96/for-web.pdf

[7] Denker, M., Suen, M., Ducasse, S., 2008. The Meta in meta-object architectures. In: Proceedings of TOOLS EUROPE 2008. Vol. 11 of LNBIP. Springer-Verlag, pp. 218–237.
URL http://rmod.lille.inria.fr/archives/papers/Denk08b-Tools08-MetaContext.pdf

[8] Dittrich, K., Gatziu, S., Geppert, A., 1995. The active database management system manifesto: A rulebase of ADBMS features. In: Sellis, T. (Ed.), Rules in Database Systems. Vol. 985 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–17.
URL http://dx.doi.org/10.1007/3-540-60365-4_116

[9] Ferber, J., Oct. 1989. Computational reflection in class-based object-oriented languages. In: Proceedings OOPSLA 1989, ACM SIGPLAN Notices. Vol. 24. pp. 317–326.

[10] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A., Nov. 1997. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In: Proceedings OOPSLA 1997. ACM SIGPLAN Notices, pp. 318–326.
URL http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html

[11] Kiczales, G., des Rivières, J., Bobrow, D. G., 1991. The Art of the Metaobject Protocol. MIT Press.

[12] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., 2001. An overview of AspectJ. In: Proceedings ECOOP 2001. No. 2072 of LNCS. Springer Verlag, pp. 327–353.

[13] Maes, P., Dec. 1987. Concepts and experiments in computational reflection. In: Proceedings OOPSLA 1987, ACM SIGPLAN Notices, pp. 147–155.

[14] Maes, P., 1988. Issues in computational reflection. In: P. Maes, D. Nardi. (Ed.), Meta-Level Architectures and Reflection. Elsevier Science Publishers B.V. (North-Holland), pp. 21–35.

[15] Marr, S., Seaton, C., Ducasse, S., Jun. 2015. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. SIGPLAN Not. 50 (6), 545–554.
URL http://doi.acm.org/10.1145/2813885.2737963

[16] Miranda, E., 2008. Cog blog. Speeding up Croquet and Squeak with a new open-source VM from Qwaq.
URL http://www.mirandabanda.org/cogblog/

[17] Mostinckx, S., Van Cutsem, T., Timbermont, S., Gonzalez Boix, E., Tanter, E., De Meuter, W., May 2009. Mirror-based reflection in AmbientTalk. Software Practice and Experience, 661–699.

[18] Papoulias, N., Dec. 2013. Remote debugging and reflection in resource constrained devices. Thèse, Université des Sciences et Technologies de Lille - Lille I.
URL http://tel.archives-ouvertes.fr/tel-00932796

[19] Papoulias, N., Denker, M., Ducasse, S., Fabresse, L., 2015. Reifying the reflectogram: Towards explicit control for implicit reflection. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. SAC '15. ACM, New York, NY, USA, pp. 1978–1985.
URL http://doi.acm.org/10.1145/2695664.2695883

[20] Toledo, R., Leger, P., Tanter, E., 2010. AspectScript: Expressive aspects for the Web. In: Proceedings of the 9th International Conference on Aspect-Oriented Software Development. AOSD '10. ACM, New York, NY, USA, pp. 13–24

[21] Redmond, B., Cahill, V., 2000. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In: Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures.

[22] Redmond, B., Cahill, V., 2002. Supporting unanticipated dynamic adaptation of application behaviour. In: ECOOP 2002. Vol. 2374 of LNCS. Springer-Verlag, pp. 205–230.

[23] Ressia, J., 2012. Object-centric reflection. Ph.D. thesis, Institut fur Informatik und angewandte Mathematik. University of Bern.

[24] Röthlisberger, D., Denker, M., Tanter, É., 2007. Unanticipated partial behavioral reflection. In: Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006). Vol. 4406 of LNCS. Springer, pp. 47–65.
URL http://rmod.lille.inria.fr/archives/papers/Roet07b-ISC06-UPBReflection.pdf

[25] Röthlisberger, D., Denker, M., Tanter, É., Jul. 2008. Unanticipated partial behavioral reflection: Adapting applications at runtime. Journal of Computer Languages, Systems and Structures 34 (2-3), 46–65.
URL http://rmod.lille.inria.fr/archives/papers/Roet08a-COMLAN-UPBReflectionJournal.pdf

[26] Smith, B. C., 1982. Reflection and semantics in a procedural programming language. Ph.D. thesis, MIT.

[27] Tanter, E., 2010. Execution levels for aspect-oriented programming. In: Proceedings of the 9th International Conference on Aspect-Oriented Software Development. AOSD '10. ACM, New York, NY, USA, pp. 37–48.
URL http://doi.acm.org/10.1145/1739230.1739236

[28] Tanter, E., Figueroa, I., Tabareau, N., Feb. 2014. Execution levels for aspect-oriented programming: Design, semantics, implementations and applications. Science of Computer Programming 80, 311–342.
URL http://dx.doi.org/10.1016/j.scico.2013.09.002

[29] Tanter, É., Noyé, J., Sep. 2005. A versatile kernel for multi-language AOP. In: Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on

Generative Programming and Component Engineering (GPCE 2005). Vol. 3676 of LNCS. Tallin, Estonia.

[30] Tanter, É., Noyé, J., Caromel, D., Cointe, P., Nov. 2003. Partial behavioral reflection: Spatial and temporal selection of reification. In: Proceedings of OOPSLA 2003, ACM SIGPLAN Notices. pp. 27–46.
URL http://www.dcc.uchile.cl/~etanter/research/publi/2003/tanter-oopsla03.pdf

[31] Wand, M., Friedman, D., 1988. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In: P. Maes., D. Nardi, (Eds.), Meta-level Architectures and Reflection. pp. 111–134.

[32] Zhuang, Y., Chiba, S., 2013. Method slots: Supporting methods, events, and advices by a single language construct. In: Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development. AOSD '13. ACM, New York, NY, USA, pp. 197–208.

[33] M.L. Gassanenko. Context-oriented programming. In *euroForth'98*, Schloss Dagstuhl, Germany, April 1998.

[34] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In: Proceedings of OOPSLA 1993, ACM SIGPLAN Notices, volume 28, pages 411–428.

[35] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.