Reifying the Reflectogram
Nikolaos Papoulias, Marcus Denker, Stéphane Ducasse, Luc Fabresse

To cite this version:

HAL Id: hal-01098596
https://hal.inria.fr/hal-01098596
Submitted on 6 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Towards Explicit Control for Implicit Reflection

N. Papoulias
RMoD, Inria Lille Nord Europe
http://rmod.lille.inria.fr
npapoylias@gmail.com

M. Denker
RMoD, Inria Lille Nord Europe
http://rmod.lille.inria.fr
marcus.denker@inria.fr

S. Ducasse
RMoD, Inria Lille Nord Europe
http://rmod.lille.inria.fr
stephane.ducasse@inria.fr

L. Fabresse
Mines Telecom Institute, Douai
http://www2.mines-douai.fr/
luc.fabresse@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 run-time 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 argue that the reification of the descriptive notion of the reflectogram, can unify the control of meta-level execution in all these five dimensions. We present a model for the reification of the reflectogram and validate our approach through a prototype implementation in the Pharo programming environment. Finally we detail a case-study on run-time tracing illustrating our approach.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features

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 [Sim82]). In OO reflective systems, reflection is concretized using a MOP (Meta-Object Protocol) [Karp91]. A meta-object is a regular object that describes, reflects or defines the behavior of a notion of the language in question [Mac87]. 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 [CCL00] are used both for implementing language extensions such as AOP frameworks [TN05] and for supporting new programming tools and methodologies such as object-centric debugging [Res12] and message based profiling [Ber11].

Yet controlling the run-time behavior of reflection introduces several challenges such as computational overhead [Mac88], the possibility of meta-recursion [CKL96] [DSD08] and an unclean separation of concerns between base and meta-level [BR04]. These problems arise mainly when implicit reflection (i.e., reflection that is activated implicitly by the interpreter on pre-defined execution events [Mac88]) alters the semantics of a running process in ways that lead to excess overhead or inconsistencies.

Implicit reflection operates similarly to an Event-Condition-Action model [DGG95] [TNCC03]. 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. In its most general form registration of such events can take the following form (code presented in Smalltalk syntax):

```
<table>
<thead>
<tr>
<th>Implicit Reflection Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>MsgSend</td>
</tr>
<tr>
<td>do: [messageCounter := messageCounter + 1]</td>
</tr>
</tbody>
</table>
```

In essence for every invocation of an Event by the interpreter (such as a message send) if a predefined Condition for that event is met (e.g., a counting flag is set), a meta-level Action is implicitly evaluated (e.g., a message counter is incremented).

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 argue that the reification of the descriptive notion of the reflectogram [TNCC03] can unify the control of meta-level execution in all these five dimensions. The idea of reflectogram (seen in Figure 3) was proposed by Tanter et al. 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 a model for the reification of the reflectogram and validate our approach through a prototype implementation in the Pharo programming environment. 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 paper are the following:
The presentation of different dimensions of meta-control that have been previously treated separately in literature.

2. DIMENSIONS OF META-CONTROL

2.1 Temporal Control

We refer to the temporal control of implicit reflection as the ability of run-time installation, activation, de-activation and removal of reflective facilities. Temporal control allows a programmer to define when a semantic event will actually be reified by controlling the time of its activation. In essence setting up meta-actions for semantic 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 Script 1 are redundant since meta-actions can be installed, enabled, disabled or removed during execution:

Script 2: Temporal Control Example

```
1. SomeClass
2. on: MsgSend
3. when: [ countingFlag = true ]
4. 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 — provide even finer control over what is reified. This is accomplished by targeting single operations in a sub-method level or even restricting reifications to specific objects over particular executions as in the case of Bifrost. With spatial control unnecessary jumps to the meta-level (e.g., for classes other than SomeClass) can be avoided resulting to 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 semantic event as exemplified by Tanter et al., but also in works related to method slots and wrappers. For example user-defined actions can be triggered before or after a semantic event or even totally replace the default semantic action.

Script 4: Placement Control

```
1. SomeClass
2. on: MsgSend
3. when: [ countingFlag = true ]
4. do: [ messageCounter := messageCounter + 1 ]
```

This is shown on lines 4 & 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 in the base-level code of SomeClass. While the second action (line 5) is our counter increment example registered to be performed after the semantic 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 such as the one of Reflex.

2.4 Level Control

Level control refers to the ability of assigning different reflective behavior to different meta-levels of a reflective tower. 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 meta-level actions are not carefully coded. In this case the problem of infinite meta-recursion occurs.

A simple case of infinite meta-recursion illustrating the problem is given on Script 5. On lines 1 to 3 of Script 1 we register (through the message #on:do:) a callback action for the MessageReceived event (line 2) of the instance anObject (line 1). Essentially we want the block close in 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, resulting in an infinite recursion.

```
1. anObject
2. #on:do:
3. [ Transcript show: 'A message was send from SomeClass' ]
4. [ messageCounter := messageCounter + 1 ]
```
Denker et al. [DSD08] first proposed a local 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):

Script 6: Code execution with a metaContext [DSD08]

```
[ ... 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 executions levels has been proposed [Tan10, TFT14]. 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 and AspectJ.

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

Identity control is the ability to distinguish between the receiver of a reflective message and the targeted object of a reflective operation. This distinction was investigated by Bracha and Ungar through Mirrors [BU04] but has been studied in different contexts as well [Fer89, Caz03]. AmbientTalk [MVC19] was the first mirror-based implementation specifically targeting implicit reflection.

Identity control can prove useful in situations as the one depicted in Figure 2. In Figure 2 anObjectInspector 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 instance variable access for aPersistentObject have changed through reflection to synchronize its state with an external data storage. Let us now assume that this was achieved by instructing the compiler to transform each read and write access of instance variables in the source code to 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 which 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 decomposed from 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. What has essentially changed here between Ob-

![Figure 2: Inspecting an intercessed object](image)

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. REIFYING THE REFLECTOGRAM

The notion of reflectogram was introduced by Tanter et al. [TNCC03] as a conceptual illustration to describe meta-level behavior to human readers: [...] A reflectogram illustrates the control flow between the base level and the metalevel during execution.

For example in the left part of Figure 3 we see a diagram from Tanter and al. describing spatial control and partial reflection, while in the bottom part of Figure 3 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 [DSD08] (right part of Figure 2) describing level control.

Given the reflectogram’s versatility for describing meta-level behavior in this section we propose its reification as a programmable entity of the meta-level. More precisely we propose its reification as an explicit meta-object that is passed as an argument at runtime to conditions & 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.

3.1 Reifying the Reflectogram

Our proposal (shown in Figure 4) 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 reified reflectogram. The relationship is one to many in the sense that a single action or condition can be registered for multiple objects but upon each invocation only the reflectogram corresponding to the particular object that triggered the event will be passed as an argument. Besides holding a reference to the targetObject, the reflectogram should provide meta-information (our reifications slot) for the currently triggered event which — depending on the implementation — can be used to parametrize conditions and actions at runtime.

**Event Registration.**

The reflectogram controls the spatial dimension of implicit reflection at runtime through the methods #on:when:do: and #on:for:when:do:, with the latter being 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:
Script 7: Registering events with the reflectogram

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

As seen on lines 4 and 5 of Script 7, conditions and actions in our model receive an argument which describes and controls the "shape" of the reflectogram for each meta-level jump of a particular object. The class-side method `#on:for:when:do:` is used for the initial registration of an event, while its instance-side counterparts `#on:when:do:` provides the same functionality from within the meta-level — as a convenience — for the specific object that triggered an event.

3.2 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 from within the meta-level. Implementors can choose to provide static counterparts for convenience (such as `#enableFor`, `#disableFor` etc.).

**Spatial Protocol.** Methods `#on:when:do:`, `#on:for:when:do:` 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 action of the base-level from within the meta-level thus implicitly defining which meta-level statements will be executed before and which after the actual semantic 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 or condition as in the meta-level tower model. Process meta-level returns the process-wide meta-level height, while object meta-level returns the height 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 reflectogram 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 8 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 8: Solving the meta-recursion problem with the reflectogram

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

Lines 1 to 3 of Script 8 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 reflectogram is explicitly disabled thus temporarily allowing message sends to be received by `anObject` without interception. On line 7 the message `incrementMessageCounter` is sent to `anObject` without
resulting in an infinite recursion since the reflectogram 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 reflectogram is re-enabled before returning control to the base-level, as to be able to intercept further message sends to anObject.

4. THE REFLECTOGRAM IN ACTION

This section presents the implementation of a non-trivial tracing framework where the code that will be traced is not a priori known (i.e., is unanticipated) but is being instrumented on-the-fly at runtime. Message-based profiling [Ber11] 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.

Figure 5 shows the core classes of our tracing framework which include:

- **CallGraph.** The entry point of the output callgraph of our tracing process.
- **CallGraphNode.** Individual nodes of the output callgraph holding the actual meta-information that have been traced. For our framework these meta-information include: 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 callgraph 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 9) which are bound to traced objects at runtime. These callbacks then delegate meta-level control to methods #inTracingScope, #addGraphNode, #executeNode: and #return: respectively (Script 10).

As seen in lines 2,8,9 and 10 of Script 9 since the reflectogram is reified as a first class entity it can be passed as an argument beyond the scope of conditions or meta-actions themselves. In line 2 the reflectogram is passed to #inTracingScope: method to determine if the meta-event that triggered the conditionCallback happened inside our tracing process or not. In line 8 it is passed to #addGraphNode: to gather the meta-information needed to update the callgraph. Then on line 9 it is passed to #executeNode: to perform the default base-level action and gather its return value. Finally on line 10 it is passed to the #return: method which resets both the callgraph and the reflectogram appropriately for further processing.

**Script 9: Tracing Callbacks**
On Script 10 we see these delegate methods in more detail:

**Script 10: Meta-control methods using the Reflectogram**

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#inTracingScope</td>
<td>Reflectogram reification process.</td>
</tr>
<tr>
<td>#addGraphNode</td>
<td>Reflectogram reification message selector</td>
</tr>
<tr>
<td>#executeNode</td>
<td>New call trace forClass: Reflectogram</td>
</tr>
</tbody>
</table>
```

Method #inTracingScope: (lines 1 to 4) the level protocol of the reflectogram is used (line 4) in order to determine whether we are intercepting a method call that originated from our tracing process' base-level (processMetaLevel = 1). If not #inTracingProcess: will return false and the corresponding meta-action (lines 5 through 9 on Script 9) will not be invoked.

Method #addGraphNode: (lines 5 through 14) the reification slot of the reflectogram is used in order to gather meta-information about the intercepted call and update the callgraph. 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 run-time via the spatial protocol (lines 19 to 23) then on lines 24 to 26 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 28 to 30) is the equivalent of a post-action, we update the callgraph (to point to the node that we have previously added) and re-enable the reflectogram (line 30) for our traced object through the temporal protocol. The reflectogram had been previously disabled for convenience (in order to avoid unnecessary meta-jumps) on the beginning of the meta-action callback (line 7, Script 9).

5. IMPLEMENTATION

The prototype implementation of our model for the reflectogram (as described in Section[3]) as well as the case-study (detailed on Section[4] are part of a dedicated virtual machine targeting the Pharo platform, the metaStackVM[1][Pap13]. We chose to implement our prototype through a virtual-machine extension (rather through some other form of instrumentation, for e.g., byte-code instrumentation) since vm-support serves better the instrumentation needs of run-time entities (for e.g., terminal instances, as opposed to static entities such as classes or methods).

In order to evaluate our solution we performed a micro-benchmark to compare the overhead introduced to normal execution with and without the reflectogram reification. The benchmark is based on Taner [TNCC03] and measures the slowdown introduced for one million messages sent to a test object when a) no instrumentation is present b) instrumentation is loaded but is disabled for this specific object c) instrumentation is enabled on the test object d) instrumentation is enabled on the test object and its reflectogram is being reified.

As we see in Table[1] when instrumentation is loaded to the environment but the benchmark object is not being instrumented, there is no additional overhead compared to the standard VM (with no instrumentation). This is important for practical reasons so as to avoid slowing down the whole system when instrumenting only a part of it [TNCC03], For example implicit reflection on the metaStackVM introduces a 9.27x overhead but only for the benchmarked object, outperforming other solutions for the same platform [Pap13]. Finally adding the reification of the reflectogram to the metaStackVM introduces a 1.37x slowdown compared to implicit reflection without such reification. We believe that the added benefit of fine-grained meta-level control using the reflectogram outweighs this additional slowdown especially when it is only introduced for objects being instrumented.

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>No instrumentation</td>
<td>1x</td>
</tr>
<tr>
<td>Disabled instrumentation</td>
<td>1x</td>
</tr>
<tr>
<td>Enabled instrumentation</td>
<td>9.27x</td>
</tr>
<tr>
<td>Reflectogram Reification</td>
<td>12.71x</td>
</tr>
</tbody>
</table>

Table 1: Instrumentation Benchmark

6. RELATED WORK - COMPARISON

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

While Iguana/J [RC00] [RC02] was the first to introduce unanticipated changes (temporal control) and spatial control for the Java platform: the metaStackVM[1][Pap13]. We chose to implement our prototype through a virtual-machine extension (rather through some other form of instrumentation, for e.g., byte-code instrumentation) since vm-support serves better the instrumentation needs of run-time entities (for e.g., terminal instances, as opposed to static entities such as classes or methods).

1. http://ss3.gemstone.com/ss/mSVM.html
platform, it was Reflex [TNCC03] that introduced partial behavioral reflection to literature (supporting spatial, temporal and placement control). On the other hand due to implementation constrains Reflex did not allow for dynamic definition of meta-level behavior at run-time as did Iguana/J. These three types of control (placement, spatial & temporal) were first available during run-time in Gepetto for Smalltalk [RDT07]. With later extensions to the Gepetto implementation covering level control [DS08]. Bifrost added an object-centric model to run-time reflection expanding spatial control to execution runs [Res12]. AmbientTalk [MVCT+09] 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 mainly comparable with Gepetto [RDT07] taking into account its later extension for level control [DS08].

Aspect-Oriented Programming.

As discussed in Section 2 the problems we 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, advices and join points. Moreover the recent proposal of executions levels [Tan10] solves the equivalent problem of meta-recursion by avoiding aspects loops.

Limitations.

From a model perspective our solution presents some limitations compared to Reflex or Gepetto’s model which are focused on extensibility. These models introduce abstractions (such as links & 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). 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 reification 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 an extension on top of them.

7. 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. It proposes a model for the reification of a previously descriptive notion — that of the reflectogram [TNCC03] — arguing that such reification can unify the control of meta-level execution in all five dimensions. We presented a prototype implementation of this reification in the Pharo programming environment and validated our approach through a case-study on unanticipated tracing. In terms of future work — apart from our own prototype — we would like to implement our model as an extention to other implicit reflection frameworks (such as Gepetto) and provide a formal semantic representation of the reflectogram.

8. ACKNOWLEDGMENTS

The authors would like to thank Prof. Eric Tanter for his valuable feedback regarding related literature in the context of AOP and for helping us strengthen our presentation. This work was partially sponsored by the European Smalltalk Users Group: [http://esug.org]

9. REFERENCES


URL: [http://bracha.org/mirrors.pdf]

doi:10.1007/s00446-003-0094-8]

