Towards Structural Decomposition of Reflection with Mirrors

Nick Papoulias[1,2] Noury Bouraqadi[2] Marcus Denker[1]
Stéphane Ducasse[1] Luc Fabresse[2]

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1
[2]Université Lille Nord de France, Ecole des Mines de Douai
{nick.papoulias,noury.bouraqadi,luc.fabresse}@mines-douai.fr,
{marcus.denker,stephane.ducasse}@inria.fr

Abstract

Mirrors are meta-level entities introduced to decouple reflection from the base-level system. Current mirror-based systems focus on functional decomposition of reflection. In this paper we advocate that mirrors should also address structural decomposition. Mirrors should not only be the entry points of reflective behavior but also be the storage entities of meta-information. This decomposition can help resolve issues in terms of resource constraints (e.g. embedded systems and robotics) or security. Indeed, structural decomposition enables discarding meta-information.

1. Introduction

Reflective Object Oriented languages provide a conceptual and design advantage due to their dynamic nature. This advantage often comes at the price of resource intensive solutions for our computational problems. In areas where there are resource constraints, reflective languages aid the design process, but their reflective nature is a burden upon deployment. Moreover security is an issue in the presence of reflective facilities and meta-information.

Mirrors [6] first introduced in the language Self [18] try to remedy this situation. Mirrors provide the decoupling of reflection from the base system through functional decomposition [3]. This means that reflective methods on objects are migrated to separate entities (called Mirrors) that conform to a specific behavior. This behavior can be described via interfaces or abstract classes, allowing different implementations. Thanks to this functional decomposition, it is possible to later discard the reflective behavior of a language.

On the contrary in classical implementations of reflection (as in Smalltalk) there is no clear specification as to where and how meta-information and reflective functionality is stored, can be accesed, and how exactly (semantically or otherwise) it interacts with the base system.

A clear example of this problematic situation is given in [6] and it's generality easily encompasses nearly all reflective OO languages in current use today. To illustrate this we provide and discuss the same example for Smalltalk in Figure 1.

What the code-snippet in Figure 1 illustrates is that the meta-linguistic part of the system, specifically in this example the acquisition of class and superclass references, is implemented and accessed as ad-hoc functionality inside the base system of the language. *Base and Meta level functionality is thus indistinguishably mixed in program code.*

```
myCar := Car new.
carClass := myCar class.
anotherCar := carClass new.
carSuperclass := carClass superclass.
```

Figure 1. The initial example of Bracha and Ungar in Smalltalk

Mirror based reflection deviates from this perspective by proposing a specific design pattern for meta-level access that can exhibit certain desired characteristics such as:

Encapsulation The ability of mirrors to encapsulate the reflective behavior and meta-information of objects, allowing different implementations.

Stratification The ability of mirrors to functionally decompose reflection from the base system, allowing the meta-level to be discarded when not needed.

Ontological Correspondence The ability of mirrors to describe and reflect a language in its entirety. This is a derived property that depends solely on the completeness of the mirror implementation.

Although mirrors support multiple implementations through encapsulation, there always exists some original source of meta-information in reflective languages. This original source may well reside in the base system of the language, in abstract syntax trees, in separate source files, in system dictionaries, in the object representation or inside the virtual-machine.

Since mirrors describe only the functional decomposition of reflection there is no specification for the stratification of the meta-information itself. This fact impacts the level of stratification that mirrors can offer.

In this paper, we propose a solution to this problem by means of structural decomposition for reflection. Structural decomposition is achieved by extending mirrors to be the storage entities of meta-information. We also distinguish between the high-level language meta-information and low-level meta-information from the perspective of the VM. We argue that although low-level information is the limit of stratification, it can be clearly decomposed from the base object model of a language.

In Section 2 we discuss stratification of reflection in the context of resource and deployment constraints. In Section 3 we provide a reference model for structural decomposition of reflection. Finally in Section 4, we describe and validate a prototype for our proposal.

2. Discussion on Structural Decomposition

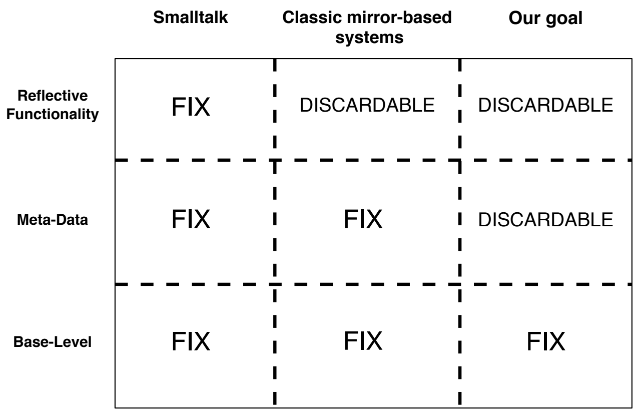
2.1 Functional and Structural Decomposition

Both in literature [6] and in state-of-the-art implementations of mirrors [4] the emphasis is given on the functional decomposition of reflection. This means that there is no specific description for the origin of meta-information or their storage point in the system. It follows that the stratification property of mirrors discussed in literature [6] does not apply to meta-data.

This is why the usual strategy for mirror instantiation is to collect the meta-information from the system as a whole (through primitives, separate source files or otherwise) [14]. Mirrors are not the original source of meta-information.

In existing mirror-based systems, one can only stratify the reflection functionality not the actual meta-data (see left part of Figure 2). This situation raises two issues. On the one hand, unused meta-data cannot be discarded to free system resources. On the other hand, undesirable access to this meta-data is possible since they still reside within the system. The impact of these facts should not be underestimated. Meta-information are widely used in reflective OO languages such as: global resolution names, names of instance variables and methods, system categories and packaging information, the code for each module, sub-module in the language, abstract syntax trees, executional, profiling and tracing information.

Structural decomposition of reflection through mirrors would help to extend the property of stratification to meta-data. Thus, one can effectively discard meta-data (see right part of Figure 2). As an analogy to this, in c-like languages one is able to discard not only debugging facilities but also meta-information [12, 17] upon deployment for efficiency reasons.



	Smalltalk	Classic mirror-based systems	Our goal
Reflective Functionality	FIX	DISCARDABLE	DISCARDABLE
Meta-Data	FIX	FIX	DISCARDABLE
Base-Level	FIX	FIX	FIX

Figure 2. Our goal, towards structural decomposition with mirrors

2.2 Low-level vs High-level Meta-information

Bracha and Ungar [6] make the distinction between low-level and high-level mirrors. We discuss this distinction in the light of structural decomposition. Although structural decomposition of all meta-information is desirable, not all meta-information should be discarded. The run-time engine does require some meta-information to actually run the base-level. We qualify such meta-information as low-level. This information usually includes: class reference, superclass reference and unique indexes for each method.

To sum up we believe that all meta-information should be decomposed and thus materialized in mirrors. Furthermore there should be a distinction as in [6] between low-level and high-level mirrors. Low-level mirrors cannot be discarded, since they encapsulate low-level meta-data. On the opposite all high-level meta-information should be materialized in high-level mirrors. Thus, it can be discarded.

3. Reference Model for the Structural Decomposition of Reflection

Aiming for simplicity and generality in our model, we chose to derive it from ObjVLisp [8]. We extended it by adding a meta-level with both an abstract and a concrete specification of mirrors. In this object model:

- Every object in the system has a Mirror, including classes and meta-classes.
- Mirrors are meta-level entities that hold all the meta-information of the respective object that they reflect on, and they are the sole provider of all reflective functionality on that object.
- All other entities in the language (including meta-classes) can provide reflective functionalities only explicitly through mirrors.

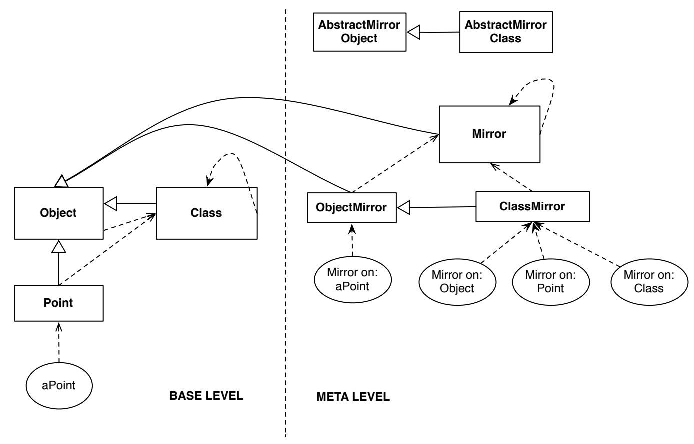


Figure 3. The MetaTalk Object Model

- Dynamic class definition can only be done through mirrors.

In our object model the base level is self-contained and can function independently. It can only instantiate terminal objects, without their meta-level counter parts. This allows the base system to operate seperately and effectively prevents dynamic addition of behavior in the absence of Mirrors.

All reflective functionality and meta-information is accessible through mirrors. This information also includes low-level meta-information which is part of an object's representation but cannot be accessed from the base level.

4. Implementation and Validation

4.1 Implementation

For validating the structural decomposition of meta-information in a mirror based reflection system we implemented the experimental class-based language MetaTalk. MetaTalk focuses on providing these characteristics to a dynamically typed OO language inspired by Smalltalk [10], Resilient (a Smalltalk descendant targeting embedded devices) [1] and ObjVLisp [8].

MetaTalk follows the guidelines of our reference object model described in Section 3. It was implemented from top to bottom (object-representation, compiler, and virtual-machine) in the open-source, smalltalk-inspired environment Pharo [2]. Our compiler relies on *PetitParser* [16]. Compilation by definition is taking place in the presence of meta-information. Only the meta-level has access to the compiler. Thus it can be discarded from the system in the absence of mirrors.The code for our open-source language prototype can be found in the SqueakSource repository[1].

[1] http://www.squeaksource.com/MetaTalk/

4.2 Validation

For the validation of structural decomposition one needs to test program execution both in the presence and in the absence of mirrors. Base level functionality should be semantically identical in both cases. On the contrary access to the meta level should only be possible in the presence of mirrors. If base level functionality is validated to be identical in both cases, this means that mirrors can be safely discarded when not needed. Moreover access to the meta-level should be validated to raise an error or an exception in the absence of mirrors.

Following this strategy for the validation of our prototype we took the following successive steps:

1. We compiled our kernel from sources, and validated its sound execution in the complete absence of the meta-level.
2. We then compiled our meta-kernel from sources, providing the system with its reflective functionality.
3. Subsequently we allowed global access to mirrors by invoking: *Baselevel reflect: true* , which is signaling the VM to permit mirrors to be pushed in the execution stack. From this point on and for the rest of the life of the system, no further compilation from sources can take place.
4. Then we dynamically created new classes, a superclass and a subclass through the meta-level (which by now is the only way to introduce new functionality to the system).
5. We tested the newly created classes for both their base and meta level functionality (via mirrors).
6. Subsequently we forbide global access to mirrors by invoking: *BaseLevel reflect: false.*
7. Finally we repeated step 5 of the process, verifying that: (a) base level functionality of the newly created classes was not by anyway altered by the absence of the meta-level, thus concluding that the meta-level could be safely discarded; (b) the subsequent attempt to access the meta-level signaled a terminal error by the vm.

Steps 4 through 7 are seen in Figures 4 and 5, respectively while the standard output generated in these steps can be found in Figure 6.

5. Related Work

The work of Bracha and Ungar in [6] discusses the functional decomposition of reflection via mirrors. We believe that we have succeeded in showing that structural decomposition is also essential for extending the property of stratification to meta-data.

The work of Lorenz and Vlissides [13] on pluggable-reflection, concerns reflection as a uniform interface in the presence of multiple sources of meta-information. It is re-

```
newClass := ((Mirror on: Object) subClass: 'PointX' instanceVariableNames: {'x' . 'y'.}) baseObject.

(Mirror on: newClass) atMethod: 'initialize' put: 'x := 0. y := 0.'.
(Mirror on: newClass) atMethod: 'x' put: '^ x.'.
(Mirror on: newClass) atMethod: 'y' put: '^ y.'.
(Mirror on: newClass) atMethod: 'x: aNumber' put: 'x := aNumber asNumber.'.
(Mirror on: newClass) atMethod: 'y: aNumber' put: 'y := aNumber asNumber.'.
(Mirror on: newClass) atMethod: 'asString' put: '^ x asString , ''@'' , y asString.'.

newSubClass := ((Mirror on: newClass) subClass: 'Point3DX' instanceVariableNames: 'z'.) baseObject.

(Mirror on: newSubClass) atMethod: 'initialize' put: 'super initialize. z := 0.'.
(Mirror on: newSubClass) atMethod: 'z' put: '^ z.'.
(Mirror on: newSubClass) atMethod: 'z: aNumber' put: 'z := aNumber.'.
(Mirror on: newSubClass) atMethod: 'asString' put: '^ super asString , ''@'' , z asString.'.
```

Figure 4. Step 4 of the validation process

lated to the encapsulation property of mirrors, allowing multiple implementations but not stratification. Their approach to the reflection interface is extralingual and does not concern the reflective system inside the language or it's decomposition from the base level.

A comprehensive comparison of mirror-based systems with other approaches for reflection in general, is given in [6].

Specifically for Smalltalk the language itself strongly couples the base and the meta-level. A declarative model for the definition of Smalltalk programs as is presented in [19]. can be used as a basis for the execution of Smalltalk programs that do not use reflection. In our work using mirrors and structural decomposition, we have shown that even in an imperative model for a language, reflection can be decoupled and stratified when it is not needed. Furthermore in our validation prototype the kernel is compiled from sources and can be used separately as a non reflective declarative system.

The Resilient platform [1] targeting embedded devices, offers remote reflection, by separating the programming from the running environment, and greatly succeeds in minimizing the deployment footprint. The reflection scheme though is not mirror based and cannot provide encapsulation. From the point of view of stratification and structural decomposition in such a scenario we believe that the deployment footprint can be reduced even more.

Implementations of mirror - based reflective sub-systems, that in general terms follow the premises of [6] already

```
'Validation process...' print.

'Base level functionality, in the presense of mirrors:' print.
p3D := newSubClass new.
p3D z: 30.
p3D x: 1.
p3D print.

'Meta level functionality, in the presense of mirrors:' print.
p3D := newSubClass new.
(Mirror on: p3D) perform: 'z:' withArguments: 30..
(Mirror on: p3D) perform: 'x:' withArguments: 1..
(Mirror on: p3D) print.

BaseLevel reflect: false.
'Base level functionality, in the absense of mirrors:' print.

p3D := newSubClass new.
p3D z: 30.
p3D x: 1.
p3D print.

'Meta level functionality, in the absense of mirrors -- should signal an error by the vm.' print.
p3D := newSubClass new.
(Mirror on: p3D) perform: 'z:' withArguments: 30..
(Mirror on: p3D) perform: 'x:' withArguments: 1..
(Mirror on: p3D) print.
```

Figure 5. Steps 5, 6 and 7 of the validation process.

exist in languages as diverse as Java [11], Self [14] , StrongTalk [5], Newspeak [4] and AmbientTalk [15] with *on-going* smaller or larger efforts to implement them for C++ [7], Scala [9], Javascript [20] and possibly other platforms.

From the point of view of structural decomposition, and although our focus is on dynamic OO languages, the ongoing effort of the C++ community for mirror - based reflection, presents some interest. The static nature of the language suggests that structural decomposition in these languages should indeed be possible.

But, it is exactly this nature of the language that forces the implementation of reflective facilities to resort to adhoc mechanisms for generating the meta-data. In the case of C++ these come in the form of explicit registration macros for each and every name-space, type or class defined. Hinting that again structural decomposition and stratification of the reflective facilities could come only in the form of recompilation without the supporting libraries and macros.

6. Conclusion and Future Work

We have raised the question of structural decomposition of reflection and meta-information, in the context of mirror-

```
MetaTalk>>> Validation process...

MetaTalk>>> Base level functionality, in the presense of mirrors:
MetaTalk>>> 1@0@30

MetaTalk>>> Meta level functionality, in the presense of mirrors:
MetaTalk>>> Mirror on: a Point3DX 1@0@30

MetaTalk>>> Base level functionality, in the absense of mirrors:
MetaTalk>>> 1@0@30

MetaTalk>>> Meta level functionality, in the absense of mirrors -
- should signal an error by the vm.
...
<'metatalk-vm-exception: meta level access is disabled'>
```

Figure 6. Standard output generated from steps 5 to 7 of the validation process.

based systems. We showed that the property of stratification for mirrors, can be weak if structural decomposition is not taken into account. We provided a solution with a reference model where mirrors are the initial source of meta-information. Finally we validated this solution through a prototype supporting both functional and structural decomposition of reflection.

In terms of future work, we would like to provide further validation and metrics for structural decomposition. We want to give a more detailed specification of the system for implementors. Furthermore we would like to advance our prototype towards ontological correspondence and examine the role of structural decomposition in the context of behavioral reflection [15].

References

[1] Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus Marius Hansen, and Mads Torgersen. Design, implementation, and evaluation of the resilient smalltalk embedded platform. In *Proceedings of ESUG International Smalltalk Conference 2004*, September 2004.

[2] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[3] Gilad Bracha. Linguistic reflection via mirrors (talk at hpi potsdam). http://bracha.org/Site/Talks.html, 2010.

[4] Gilad Bracha. Newspeak programming language draft specification version 0.06. http://bracha.org/newspeak-spec.pdf, 2010.

[5] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, October 1993.

[6] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[7] Matus Chochlik. Mirror c++ reflection utilities. http://kifri.fri.uniza.sk/ chochlik/mirror-lib/html/, 2011.

[8] Pierre Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, December 1987.

[9] Yohann Coppel. Reflecting scala. http://lamp.epfl.ch/teaching/projects/archive/coppel_report.pdf, 2008.

[10] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[11] Sun microsystems, java platform debugger architecture. http://java.sun.com/products/jpda/.

[12] David MacKenzie Julia Menapace, Jim Kingdon. The "stabs" debug format, 2004.

[13] David H. Lorenz and John Vlissides. Pluggable reflection: decoupling meta-interface and implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, Washington, DC, USA, 2003. IEEE Computer Society.

[14] Jacques Malenfant, Christophe Dony, and Pierre Cointe. Behavioral Reflection in a prototype-based language. In A. Yonezawa and B. Smith, editors, *Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures*, pages 143–153, Tokyo, November 1992. RISE and IPA(Japan) + ACM SIGPLAN.

[15] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings the ACM Dynamic Languages Symposium (DLS 2007)*, October 2007.

[16] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[17] Stan Shebs Richard Stallman, Roland Pesch. *Debugging with GDB*. Gnu Press, 2003.

[18] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for self. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 303–330, Aarhus, Denmark, August 1995. Springer-Verlag.

[19] Allen Wirfs-Brock. A declarative model for defining smalltalk programs. invited talk at oopsla 96. http://www.smalltalksystems.com/publications/_awss97/SSDCL1.HTM, 1996.

[20] Allen Wirfs-Brock. A prototype mirrors-based refection system for javascript. https://github.com/allenwb/jsmirrors, 2010.