Run-Time Evolution through Explicit Meta-Objects[⋆]

Jorge Ressia, Lukas Renggli, Tudor Gîrba and Oscar Nierstrasz

Software Composition Group
University of Bern
Switzerland
`http://scg.unibe.ch`

Abstract. Software must be constantly adapted due to evolving domain knowledge and unanticipated requirements changes. To adapt a system at run-time we need to reflect on its structure and its behavior. Object-oriented languages introduced reflection to deal with this issue, however, no reflective approach up to now has tried to provide a unified solution to both structural and behavioral reflection. This paper describes ALBEDO[1], a unified approach to structural and behavioral reflection. ALBEDO is a model of fined-grained unanticipated dynamic structural and behavioral adaptation. Instead of providing reflective capabilities as an external mechanism we integrate them deeply in the environment. We show how explicit meta-objects allow us to provide a range of reflective features and thereby evolve both application models and environments at run-time.

1 Introduction

Classical software development plays out like a finite game with fixed rules and boundaries. However, evolving software systems are rather an infinite game without fixed rules or boundaries [2]. Large systems not only evolve in the development phase, but also at run-time. Development itself is part of the infinite game of the system. The system may continue to evolve while running.

To enable change at run-time, a system must be self-aware and be able to fully reflect on itself [14]. A reflective system provides a description of itself that can be queried (introspection) and changed (intercession) from within. Consequently the system can reflect on itself and can change its structure and behavior. However, this is not enough to support full run-time evolution of models. To evolve a model at run-time it is necessary to access the dynamic representation of a program, that is, the operational execution of the program. This is called behavioral reflection, pioneered by Smith [18, 19] in the context of Lisp. A reification

⋆ Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)

1 In astronomy, albedo is the proportion of the incident light or radiation that is reflected by a surface, typically that of a planet or moon [The Oxford Dictionary of English]

not related to the structure of the language might be required, for example, a message send.

As an example, let us look at a financial model and how to make it available to an external audit system. Such a model typically contains sensitive information and certain properties should not be accessible and modifiable by everybody. We therefore need a representation of the model that has the same behavior except for certain restricted accessors. To achieve this, we need to change the model and integrate contextual security checks. However, such a transformation is not straightforward and might add unnecessary complexity to the application code. What we need is to modify the system so that security concerns can be quickly changed to adapt to new requirements. A reflective system where we can express from the inside these structural constraints to access or modify certain state will solve our problem.

It can be argued that this problem can be solved by using Aspect-Oriented Programming (AOP) [10]. AOP provides a general model for modularizing cross cutting concerns. Join points define points in the execution of a program that trigger the execution of additional cross-cutting code called advice. Join points can be defined on the run-time model (i.e., dependent on control flow). Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system. Modeling the changes in the financial problem through AOP will deliver a set of join points which have a larger semantic gap to the actual requirement than using the reifications of the system. The reflective approach is better suited for continuous evolution since join points are harder to reflect upon.

The first object-oriented language to propose a clean reflection approach was Smalltalk-80 [7] by introducing meta-classes. ObjVlisp [3] and Classtalk [1] followed this line of research. Meta-classes define the internal structure and behavior of a class. There can be only one meta-class per class. There is a second approach to reflection in object-oriented languages introduced by Pattie Maes in 3-KRS [11, 12]. This approach states that each object is related to one meta-object. A meta-object specifies the structure of the object and how this object handles messages. Maes also benefited from the work of Kiczales *et al.* [9] on meta-object protocols (MOPs) over the CLOS language, an object-oriented extension of Lisp. The protocol of a meta-object encodes the behavior of the language. If the MOP is changed the language is changed as well.

To solve the financial model security issue we can either remove methods from classes or from meta-objects. Depending on the reflection approach we will be affecting one object or all instances of a class. But the key point to highlight is that we are dealing with structural elements only.

Regardless of the differences between the reflection approaches, they are built on structural reifications of the concepts that form the language (classes, objects, methods, instance variables, etc.), however, this structural approach does not solve all reflection problems. Assume there is a new requirement which says that every time the interest rate of a financial instrument is changed a message should be sent to the central audit system. Thinking in terms of classes, objects and methods is not enough to solve this requirement.

Following Smith's work on behavioral reflection McAffer [13] developed CodA, a system that approaches reflection from the point of view of operational decomposition. He proposed to separate the description of the computational behavior of an object from that of its base language structure. This approach divides the execution of an object into basic operations (*e.g.,* message send and receive, state access, object creation and deletion, etc.).

Iguana [8] further improved this approach by providing a mechanism, known as fine-grained MOP, to allow multiple reflective object models to coexist. For example, one object can have a distributed object model while other objects in the system have a centralized object model. Although structural approaches have the advantage of organizing the meta-level in terms of concepts known to the user like classes and methods, it is hard to integrate new concepts that are not represented in the language. This is important since it allows the language to evolve beyond its predefined structure. This approach provides a way of integrating reified concepts that are not originally part of the language through its operations. There are seven reification categories: object creation and deletion; message sends, receive and dispatch; and state read and write. Iguana was later ported to Java [15, 16]

The audit requirement can be solved by specifying that a message is sent to the central audit system every time a state write event related to the financial instrument interest rate is issued.

Up until now, no approach has tried to provide a unified solution to these two domains of reflection: structural and behavioral. In this paper we present Albedo, a model of fine-grained unanticipated dynamic structural and behavioral adaptation. Instead of providing reflective capabilities as an external mechanism we integrate them deeply in the environment. We show, by modeling meta-objects explicitly, how we can extend the environment from both a structural and behavioral standpoint.

The most relevant characteristics of Albedo are:

– It provides a meta-object approach to reflection.
– The meta level can be organized with language concepts and operational decomposition.
– The fine-grained MOP allows us to control the scope of the change.

Outline. Section 2 explains how the model behind the Albedo environment solves both structural and behavioral requirements. In Section 3 we present the internal implementation of our solution in the context of Smalltalk. Section 4 presents a couple of structural and behavioral reifications and how they are modeled with meta-objects. In Section 5 we summarize the paper and discuss future work.

2 ALBEDO Approach

The ALBEDO mechanism is placed at the center of the system where every language construct is expressed in terms of an ALBEDO meta-object. In this section we show the two building blocks of ALBEDO and how they can solve the previously presented problems.

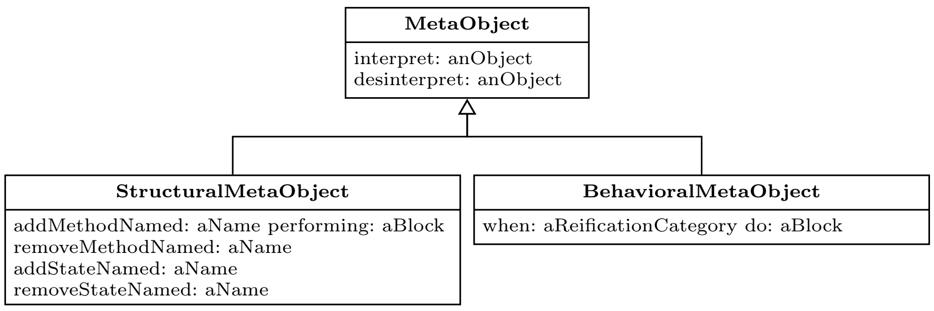

Fig. 1. Meta-Objects class diagram.

Figure 1 shows a simplified meta-object class hierarchy diagram. The meta-object abstraction is reified and it has the responsibility to adapt objects. The idea is that meta-objects specify how an object meta-level should work.

2.1 Structural Meta-object

The StructuralMetaObject abstraction reifies the meta-object responsible for modeling the structures of a program. An object-oriented program's canonical structures are objects and messages. In class-based languages, classes have too many responsibilities. They are used to model an abstraction, provide an instance creation mechanism, define the messages the instances know how to answer, and provide a template that subclasses can extend. In prototype-based languages the idea of generalization does not exist and new abstractions are built by delegating to other abstractions or objects. Traits are an example of a structural meta-object, created for sharing behavior. A trait [17] is a composable unit of behavior that can be shared among objects. If several objects share a trait then they all will be able to understand the messages defined in the trait. The StructuralMetaObject abstraction provides the means to define meta-objects like classes, prototypes and traits. New structural abstractions can be defined to fulfill some specific requirement. For example we can define the concept of traits as we discuss in Section 4.

A StructuralMetaObject acts on the basic structural units of an object-oriented language which are messages, objects and the object state. The responsibilities of a StructuralMetaObject are:

– *Adding a method.* A new method is added to the object. A name and the source code is provided. When the object receives the message it executes the compiled source code. The source code compilation is performed when the object is associated with a meta-object. If there is a compilation error the meta-object association is rolled back.
– *Removing a method.* The adapted object will not understand a particular message any more.
– *Replacing a method.* The method will have another behavior. Source code or a closure can be provided.
– *Adding state.* The addition of new state to an object allows the user to add methods that use that state.
– *Removing state.* Specific state is removed.

These responsibilities are modeled as meta-actions. A StructuralMetaObject can be defined as a particular set of these meta-actions that structurally adapt a particular object. When an object is associated to a meta-object these actions are executed and the object is adapted accordingly.

Let us consider the financial domain and pick a financial instrument which has to be audited by an external company. The auditors do not want to see a report, they want to see the real system and interact with it. This financial instrument is still active and should become due in some months. Financial instruments can be renegotiated, changing the due date and recalculating the taxes under a new contract. We want to hide this kind of behavior from the audit committee to assure the integrity of the instrument.

The following code fragment shows how this scenario can be achieved with Albedo:

```
1 anImmutableBehavior := StructuralMetaObject default.
2 anImmutableBehavior removedMethodNamed: #renegotiate.
3 anImmutableBehavior boundTo: aFinancialInstrument.
```

Listing 1. Making a financial instrument immutable regarding the renegotiation behavior.

First, we get the default structural meta-object (line 1) whose responsibility is to define the allowed behavior for immutable financial instruments. Then we remove the renegotiate method from the meta object (line 2). Finally, the meta-object is associated with the financial instrument (line 3). This association triggers the adaptation of the objects thus removing renegotiate from the set of messages understood by the financial instrument. If this message is sent to the financial instrument an error is thrown, however, if this message is sent to any other financial instrument, even to another instance of the same class, the original behavior is preserved.

Structural meta-objects deal with the definition of meta-level structural reifications. How and when they are introduced at run-time is the job of the behavioral meta-object.

2.2 Behavioral Meta-object

The BehavioralMetaObject abstraction reifies the meta-object responsible for modeling the dynamic representation of a program. By dynamic representation we refer to the language's dynamic reifications. This abstraction coresponds to the work done in Iguana and later used by McAffer in Coda. As McAffer pointed out, the system is modeled as a set of operations whose occurrences *"can be thought of as events which are required for object execution"* [13].

To dynamically adapt the behavior of an object we need to describe what we would like to do and when. To specify what we would like to apply we delegate to a specific meta-object with the responsibility of managing an event. We specify when it should be adapted by using a computational event in the execution of a program, for example, the creation of an object, sending a message, *etc.* A set of canonical events models the basic operations known as *dynamic reification categories.* These are not the only reifications possible. New dynamic reifications can be defined, the only requirement being to specify when they should be triggered.

The dynamic reification categories are:

– Object creation
– Object deletion
– Message send
– Message receive
– Message dispatch
– State read
– State write

We selected these categories following the Iguana approach. With these basic categories we are capable of adapting an object's behavior regarding operations that are executed over the language building blocks.

For example, let us consider a new requirement which specifies that every time the interest rate of a financial instrument is changed, a message should be sent to the central audit system.

```
1 aMethodLookupMetaObject := BehavioralMetaObject default.
2 aMethodLookupMetaObject
3         when: (StateRead new)
4         do: [:owner :state |
5              CentralAuditSystem
6                 interestRateOf: owner
7                 changedTo: state].
8 aMethodLookupMetaObject boundTo: aFinancialInstrument.
```

Listing 2. Dynamically reifying method lookup for a financial instrument.

In line 1 the default behavioral meta-object is obtained. Lines 2–7 show the usage of reification categories to define when the meta-behavior should to happen. In this case the central audit system is informed of a change in the interest rate of an object. In line 8 the object is associated to the meta-object, and from this point on the new meta-behavior is obtained every time the interest is changed.

3 Implementation

ALBEDO is the prototype of our meta-object model approach built in Pharo Smalltalk[2]. REFLECTIVITY [4] was used for dynamic adaptation. REFLECTIVITY provides *unanticipated partial behavioral reflection* at the sub-method level, using ASTs rather than source code or bytecode as underlying model of the software. We decided to use REFLECTIVITY since *partial behavioral reflection* as pioneered by Reflex [20] is particularly well-suited for dynamic reifications because it supports a highly selective means to specify where and when to reify an abstraction in the system. *Partial behavioral reflection* offers an even more flexible approach than pure behavioral reflection. The key advantage is that it provides a means to selectively trigger reflection, only when specific, predefined events of interest occur.

The core concept of the Reflex model of partial behavioral reflection is the *link*. A link invokes messages on a meta-object at occurrences of marked operations. The attributes of a link enable further control of the exact message to be sent to the meta-object. Additionally, an activation condition can be defined for a link which determines whether or not the link is actually triggered.

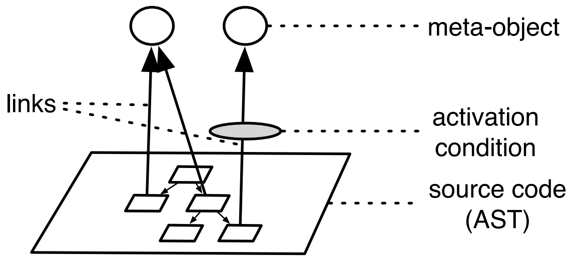

Fig. 2. The reflex model.

Links are associated with AST nodes. Subsequently, the system automatically generates new bytecodes that take the link into consideration the next time the method is executed.

REFLECTIVITY integrates itself into Pharo by using the Reflective Methods abstraction. A Reflective Method knows the AST of the method it represents. In Pharo classes are first class objects that are available to any program. They have an instance variable named `methodDict` which holds an instance of MethodDictionary – a special sub-class of Dictionary. All methods of a class are stored in its method dictionary. The VM directly uses the class objects and their method dictionary when performing message sends. Normally, only instances of `CompiledMethod` are stored in the method dictionary of a class but Pharo allow us to store any kind of object there. The VM recognizes objects that are not

[2] `http://pharo-project.org/`

instances of CompiledMethod and instead of executing bytecode the VM sends run:with:in: to the object stored in the method dictionary. When a reflective method receives this message it processes the adaptations specified in the meta-object on the AST and generates a new compiled method which is eventually executed. If no adaptation is present the reflective method caches the compiled method for performance savings. Figure 3 visualizes this relationship between a class, the method dictionary and the methods.

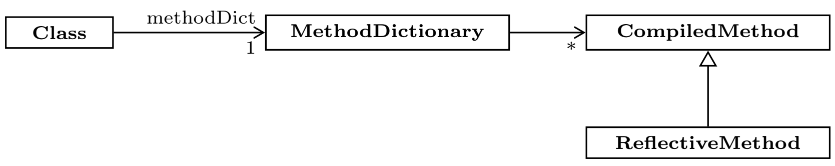

Fig. 3. Reflective Methods in Method Dictionaries

Reflectivity was conceived as an extension of the Reflex model of *Partial Behavioral Reflection* [20]. Reflex was originally realized with Java. Therefore, our approach can in be implemented in a more static mainstream language like Java. The reason for choosing Smalltalk and Reflectivity for this work is that it supports *unanticipated* use of reflection at runtime [4] and is integrated with an AST based reflective code model [5]. A Java solution would likely be more static in nature: links cannot be removed completely (as code cannot be changed at runtime) and the code model would not be as closely integrated with the runtime of the language.

4 Discussion

Albedo is not only useful for evolving systems at run-time but it is also useful for developing new language features. By modeling meta-objects explicitly we can extend the environment from both a structural and behavioral standpoint. In this section we discuss the language impact of having a system that can be extended from a structural and behavioral point of view. Next, we introduce two examples of reifying language features to show how the language itself can be modified.

4.1 Traits Example

A trait [17] is a composable unit of behavior that can be shared among objects. If several objects share a trait then they all will be able to understand the messages defined in the trait.

Let assume that we want all financial instruments to share the same behavior. For example, suppose we want to provide a common implementation for the renegotiation feature. Furthermore we do not want to impose a class hierarchy structure on all financial instruments to introduce this feature, but instead keep the possibility to assign the feature dynamically to an instrument. We can easily fulfill these needs by defining the feature as a trait, however if the host language does not implement traits we cannot introduce this feature as we would like. ALBEDO provides a mechanism to define dynamically the trait abstraction thus modifying the language model.

```
1 aTrait := StructuralMetaObject default.
2 aTrait addMethodNamed: #regenerate performing: 'self recalculateTaxes.
3                              self recalculateDates'.
4 aFinancialInstrument := aFinancialInstrumentClass new.
5 anotherFinancialInstrument := aFinancialInstrumentClass new.
6 aTrait boundTo: aFinancialInstrument.
7 aTrait boundTo: anotherFinancialInstrument
```

Listing 3. Building the trait abstraction with structural meta-objects.

First, we introduce the trait abstraction itself as a structural meta-object in line 1. The message renegotiate is defined in line 2 for this trait, its behavior is to recalculate taxes and dates. By using the existing Class abstraction defined with meta-objects we create two financial instruments in lines 4-5. The class abstraction is built using a structural meta-object and adding a message new that creates an instance of a class. The class is bound to the instance as a meta-object. Finally, we associate the trait as the meta-object to both objects thus making them capable of answering the message renegotiate.

A trait is defined as a StructuralMetaObject. However, by definition, traits should not have state. To achieve this we need to remove the possibility of adding state in the trait structural meta-object.

```
1 aTraitBehavior := StructuralMetaObject default.
2 aTraitBehavior removedMethodNamed: #addStateNamed: .
3 aTraitBehavior removedMethodNamed: #removeStateNamed: .
4 aTraitBehavior boundTo: aTrait.
```

Listing 4. Making traits stateless.

We first define another structural meta-object called TraitBehavior (line 1). This abstraction has the responsibility of defining which are the messages a trait meta-object is capable of answering. In lines 2–3 both state-related messages are removed from the trait behavior definition. Finally, in line 4 the TraitBehavior is set as the meta-object of the trait meta-object defining its responsibilities.

4.2 Method Lookup Example

The method-lookup reification defines the process that specifies which method should be executed when an object receives a message. To reify method-lookup in some languages it is necessary to perform complex computations, apply method wrappers, manipulate method dictionaries [6] or adapt byte-code.

Most languages, including Java and Smalltalk, do not reify method lookup. Being able to change the way a message is mapped to a method has many applications, for example test coverage, benchmarking and logging. Moreover, class-based languages impose a method lookup strategy that follows the class hierarchy. In some cases we need to have a strategy different from the traditional method lookup. We might not be so sure about how the class hierarchy should be organized. Here we have to provide a different strategy for matching a message to a method. For example, a financial instrument needs to reuse the behavior defined in another financial instrument which is not its super-class. We are not yet sure how the hierarchy should be reshaped, or if we should use composition instead of inheritance. So we decide to provide a new method lookup strategy to the financial instrument for checking first if the method is defined in the second financial.

To implement method lookup a two-level meta-object structure is required as shown below.

```
 1 aMethodLookupBehavior := StructuralMetaObject default.
 2 aMethodLookupBehavior addStateNamed: #strategy.
 3 aMethodLookupMetaObject := BehavioralMetaObject default.
 4 aMethodLookupMetaObject
 5         when: (MessageReceived new)
 6         do: [:receiver :message :args |
 7              strategy
 8                 resolve: message
 9                 for: receiver
10                 with: args ].
11 aMethodLookupBehavior boundTo: aMethodLookupMetaObject
12 aMethodLookupMetaObject boundTo: aFinancialInstrument.
```

Listing 5. Reifying method lookup with behavioral meta-objects.

The method lookup meta-object itself is a behavioral meta-object that reifies a message received by delegating to a predefined strategy. In lines 1–2 we define the structural meta-object and introduce a strategy instance variable. This is a meta-meta-object named aMethodLookupBehavior that specifies that the interpreted object will have a strategy instance variable. In lines 4–10 the method lookup meta-object is defined and the MessageReceived behavioral reification category is used to adapt how the method lookup should be resolved. The resolution of the lookup is delegated to the strategy active at run-time. The strategy can be changed at any point in time, thus delivering different behavior for the same object receiving the same message. Finally, aMethodLookupBehavior is associated to the meta-object of aMethodLookupMetaObject. Any object that is associated with aMethodLookupMetaObject will change the way the method lookup is resolved for that particular object.

5 Conclusion

In this paper we have introduced Albedo, a unified approach to behavioral and structural reflection. Albedo is a reflective model and environment for dynamically defined unforeseen language feature reifications. Even the most basic constructs of a language are expressed in terms of it. We have shown how different structural and behavioral meta-objects like traits and method lookup can be modeled with this environment.

What is more, we provide the means to dynamically define or modify abstractions thus eliminating the limitations of reflective models. This allows us to redefine static or run-time abstractions and manipulate their structure and behavior thus helping in the evolution of run-time models.

As future work we would like to analyze the performance impact of ALBEDO. We also would like to analyze various use cases to further validate the model. We would like to further research the composition of different meta-objects.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010) and CHOOSE, the Swiss Group for Object-Oriented Systems and Environments. We also thank Mircea Lungu and Fabrizio Perin for their feedback on earlier drafts of this paper.

References

1. Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 419–432, October 1989.
2. James P. Carse. *Finite and Infinite Games — A Vision of Life as Play and Possibility.* Ballantine Books, 1987.
3. Pierre Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, December 1987.
4. Marcus Denker. *Sub-method Structural and Behavioral Reflection.* PhD thesis, University of Bern, May 2008.
5. Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.
6. Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.
7. Adele Goldberg and Dave Robson. *Smalltalk-80: The Language.* Addison Wesley, 1989.
8. Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The Iguana approach. Technical report, AAA, 1996.
9. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.
10. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.
11. Pattie Maes. *Computational Reflection.* PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, January 1987.
12. Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.
13. Jeff McAffer. Meta-level programming with coda. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

14. Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, Adrian Lienhard, and David Röthlisberger. Change-enabled software systems. In Martin Wirsing, Jean-Pierre Banâtre, and Matthias Hölzl, editors, *Challenges for Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 64–79. Springer-Verlag, 2008.
15. Barry Redmond and Vinny Cahill. 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*, 2000.
16. Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.
17. Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks.* PhD thesis, University of Bern, February 2005.
18. Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language.* Ph.D. thesis, MIT, Cambridge, MA, 1982.
19. Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of POPL '84*, pages 23–3, 1984.
20. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.