Submission to SCP Special Issue on Experimental Software and Toolkits

Flexible Metaprogramming and AOP in Java

Éric Tanter [a] Rodolfo Toledo [a] Guillaume Pothier [a] Jacques Noyé [b]

[a] *DCC, University of Chile – Santiago, Chile*

[b] *OBASCO Project EMN/INRIA, LINA – Nantes, France*

Abstract

Advanced programming techniques such as metaprogramming and computational reflection, as well as the more recent paradigm of aspect-oriented programming (AOP), serve important objectives of software engineering such as modularization and adaptability. In this tool presentation paper, we briefly overview this area and present Reflex, a tool for flexible metaprogramming and AOP in Java. Based on a uniform model of partial reflection, Reflex provides both structural and behavioral facilities, which makes it easy to experiment with (combinations of) advanced uses of AOP and reflection without reinventing the wheel or being limited to a specific AOP language.

Key words: Metaprogramming, reflection, aspect-oriented programming, Reflex, Java

1 Introduction

Research in programming languages has been driven by the need to achieve well-modularized software respecting the principle of Separation of Concerns [1, 2]. Good modularization serves many software engineering properties such as understandability, extensibility, reusability, etc. It also helps support *adaptable* software [3, 4], since for a given concern to be adaptable (possibly dynamically) it first has to be cleanly modularized. Work on computational reflection [5, 6], metaprogramming, and more recently, Aspect-Oriented Programming (AOP) [7–9], has been a fruitful path to achieve better modularization and adaptation in many systems, such as middleware [10], concurrent

Email addresses: etanter@dcc.uchile.cl (Éric Tanter), rtoledo@dcc.uchile.cl (Rodolfo Toledo), gpothier@dcc.uchile.cl (Guillaume Pothier), noye@emn.fr (Jacques Noyé).

systems [11, 12], distributed programming [13–16], operating systems [17, 18], user interfaces [19], context-aware applications [20, 21], etc.

The Java programming language only offers a limited set of reflective abilities, which have been progressively extended as the language matured. Still, many fundamental reflective features are missing. This is why many reflective and/or aspect-oriented extensions of Java have been proposed; just to name a few: Javassist [22], for structural reflection at load time, Kava [23], for runtime behavioral reflection, and on the AOP side, AspectJ [24], the most popular Java language extension for AOP, and frameworks such as AspectWerkz [25], JAC [26], and JAsCo [27].

This paper gives an overview of Reflex, a portable Java framework for flexible metaprogramming and AOP. Reflex bridges the gap between metaprogramming and reflection on one side and AOP on the other side, and hence provides advanced users with a versatile kernel for experimenting with AOP concepts and language features [28–31]. It has been applied in concrete contexts such as concurrent systems [12], distributed systems [16, 32], and context-aware applications [21]. It is an open source project distributed under the MIT license, and the Reflex website [1] gives access to many resources, such as documentation and tutorial, subversion repository, mailing lists, and publications.

In the next section, we give a bit more background information on metaprogramming, reflection and AOP. We then present the key features of Reflex in Section 3, by exposing its underlying model and concepts. API details are not addressed here, the reader is rather referred to the Reflex tutorial on the website. We finally briefly discuss related systems in Section 4. Section 5 concludes with on-going and future work.

2 Metaprogramming and AOP

After the seminal work of Brian C. Smith on computational reflection [5, 33], and the marriage of reflection and object-oriented programming by Pattie Maes [6], many attempts have been made to apply so-called *metaobject protocols* (MOPs) [34] for achieving separation of concerns [35]. The basic idea is that the semantics of a base program is modularly extended or modified by appropriate *metaobjects*. A metaobject is given control over *reifications* of the structure or behavior of the underlying program, *i.e.* objects describing otherwise implicit elements of a program. Hence metaobjects can take care of particular *concerns* of the application, such as authentication, or invariant checking, while the base application is mostly unaware of these concerns.

[1] http://reflex.dcc.uchile.cl

This led to the fundamental issue of *metalevel engineering* [36], that is, how to organize metalevel entities in ways that are satisfying with respect to the traditional engineering principles of composability, extensibility and flexible granularity. These issues have given rise to many reflective architectures, exploring different approaches to metalevel engineering. A particularly interesting one is the *operational* decomposition proposed by McAffer [36]. McAffer distinguishes between two approaches to reflection, which consist of either starting from the base-level language structural elements (*e.g.* classes), or from the basic *operations* (message send and receive, field access, object creation, etc.) defining the computational behavior of an object. He refers to these approaches as the top-down and the bottom-up approach, respectively. One could alternatively refer to them as a structural and a behavioral approach. McAffer justifies the use of the second approach as it is more flexible in terms of granularity and makes it possible to describe a wider range of behavior models.

At the same time, work on open implementations [19] was facing the same issues of metalevel locality of change and engineering. Furthermore, Kiczales noticed that sometimes the metalevel concepts that are most natural to use actually *crosscut* the concepts at the base level [37]. This led his group to focus on this crosscutting issue and to eventually come up with the paradigm of Aspect-Oriented Programming [7] (AOP). AOP is now a very active research area [8, 9].

AOP puts forward a new kind of module called an *aspect*, which is the modular definition of a crosscutting concern. An aspect can act on a program by synchronizing with it at *join points*, usually defined as program execution points where an aspect applies, and performing its action, often called an *advice*, a term inherited from Lisp. Although the most famous join point model is the dynamic join point model, whereby a join point is simply a program execution point, which greatly ressembles the operational decomposition of McAffer, a join point model can also refer to other program properties (*e.g.* data flow graphs [7], traces [38], structural properties [39]). Individual join points are grouped together by means of *pointcuts*, which can be seen as queries on the program structure and the program execution.

3 Reflex

Reflex is a portable library that extends Java with structural and behavioral reflective facilities. We first describe the uniform model of partial reflection that lies at the heart of Reflex, before surveying the structural and behavioral facilities of Reflex. We end this section with a brief discussion of Reflex as a versatile kernel for AOP.

3.1 Uniform model of partial reflection

The underlying model of partial reflection of Reflex is that of explicit *links* binding a *cut* to an *action*. A cut specifies which program elements are of interest, the action specifies what to do on these program elements. The link is an *explicit* entity binding both, characterized by several attributes. Links are the basic unit of specification in Reflex, and can be defined either eagerly before an application starts, or dynamically while the application is running.

The cut of a link is defined via *selection predicates*, as illustrated later in this section, and the action is implemented in a *metaobject*. A metaobject can be *any* standard Java object, provided it implements the expected protocol. There are two kinds of links: *structural links* and *behavioral links*. The former are used to perform structural reflection at load time, while the latter are used to perform behavioral reflection at run time.

Links can be used to implement aspects in the AOP sense, because they do support the specification of *crosscutting* modifications to program (structure and execution). Note that our view of AOP is inherently related to metaprogramming: an aspect cut is realized by *introspection* of a program (both structure and execution), and its action consists of behavioral/structural modifications (*intercession*).

3.2 Structural facilities

A structural link binds a set of classes to a metaobject, which can both introspect and modify class definitions via a class-object structural model similar to that of Javassist [22] (Fig. 1): an `RPool` object gives access to `RClass` objects, which in turn give access to their members as `RMember` objects (either `RField`, `RMethod`, or `RConstructor`), which in turn give access to their bodies as `RExpr` objects (with a specific type for each kind of expression). These objects are *causally-connected* representations of the underlying bytecode, meaning they offer source-level abstractions to observe and manipulate bytecode.

A structural link can be used to perform any kind of structural modification to a class before it is loaded: this includes adding members to it, making a class implement a new interface, change some method signatures, etc. At this stage, modifying a method *body* is not permitted, as it would lead to serious interferences with the behavioral reflective facilities.

The cut is defined intentionally in a *class selector*, *i.e.* a predicate over `RClass` objects. For instance, the following class selector matches all classes that are *direct subclasses* of the class `Object`:

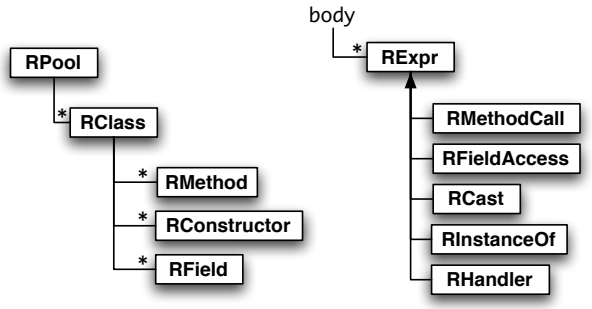

Fig. 1. Load-time structural model of Reflex.

```
ClassSelector objSubs = new ClassSelector() {
    public boolean accept(RClass aClass){
      return aClass.getSuperclass().equals(RClass.OBJECT);
  } };
```

A structural metaobject should implement the `handle(RClass)` protocol and thereby perform the desired modifications. The following piece of code shows a metaobject that adds a unique identifier to instances of the given class, hence adding a field, interface, and method to the class:

```
SMetaobject uidAdder = new SMetaobject(){
    public void handle(RClass aClass){
      aClass.addField(...); aClass.addInterface(...); aClass.addMethod(...);
  } }
```

Finally, the following statements bind the `objSubs` class selector to the `uidAdder` metaobject, and install the corresponding structural link:

```
SLink sl = Links.get(objSubs, uidAdder);
sl.install();
```

Once `install` is executed, any direct subclass of `Object` that is loaded gets transformed by `uidAdder` so that its instances will own a unique identifier field, a method to access it, and the corresponding interface.

3.3 Behavioral facilities

Behavioral reflection follows a model of partial behavioral reflection presented in [28]: the central notion is that of an explicit *link* binding a set of program points (a *hookset*) to a *metaobject*. A link is characterized by a number of attributes, among which the control attribute tells how the metaobject af-

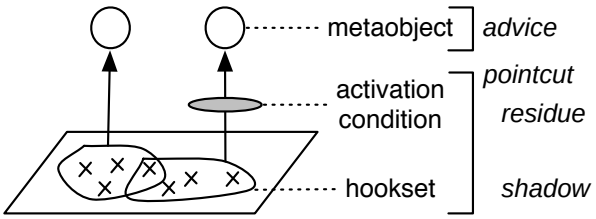

Fig. 2. The behavioral link model and correspondence to AOP concepts.

fect the base program (before, after, or around the program points), and a dynamically-evaluated activation condition makes it possible to activate and deactivate the link. Relying on a precise selection of when reification occurs, both spatially and temporally, drastically limits the cost of using reflection since jumps to the metalevel only occur when needed [28].

Furthermore, this flexible model solves the metalevel engineering problems mentioned in Sect. 2 because the metalevel decomposition is not restricted to be entity-based (*e.g.* one metaobject per object) or operation-based (*e.g.* one metaobject for message sending, one for object creation, etc.). Hence, the organization of the metalevel can rather be completely *concern-based*, *i.e.* a metaobject can realize a single concern of the application, possibly being connected to several entities and/or operations in the application. This is a first step in bridging behavioral reflection and dynamic crosscutting in AOP.

Finally, our approach promotes *open metaobject protocol (MOP) specification*, meaning that the actual protocol between the base program and metaobjects is not fixed and can be customized on a fine-grained basis (on a *per-link* basis), by means of *call descriptors*. For instance, if a metaobject only needs to be passed the name of the currently-executing method, then only that name will be passed, avoiding the overhead of full reification typically present in classical MOPs (*i.e.* implying the creation of a `MsgReceive` object which encapsulates the method object, an array of arguments, etc.).

This last point bridges the gap between behavioral reflection and AOP, since it eventually makes it possible to hide that a metaobject has something "meta": it just becomes the receiver of an intentionally-specified communication triggered by some events in program execution. Also, our experience with a concurrency model based on a simple but customized MOP confirmed that precise specification of the MOP is a great factor of performance improvement [12,30]. Using call descriptors is also better from a software design point of view, because metaobjects do not have to implement an overly-generic protocol.

Fig. 2 depicts two links, one of which is not subject to activation, along with the correspondence to the AOP concepts of the pointcut/advice model. A detailed case study of supporting the dynamic crosscutting of AspectJ in Reflex can be found in [29].

The cut of a behavioral link is defined as a *hookset*. A *primitive* hookset is specific to an operation: it defines which operation is of interest (*e.g.* object creation), and further discriminates occurrences of interest depending on the class in which they occur (by means of a class selector as in the previous section), and on their characteristics, by means of an *operation selector*, *i.e.* a predicate over the properties of reified operation occurrences. Composite hooksets can then be built using union, difference and intersection.

For instance, the following hookset matches *both* occurrences of a public method execution in objects of a subclass of `A` and accesses to fields of objects of type `A` occurring *outside* objects of `A`:

```
final ClassSelector a = new NameCS("A", true);

Hookset mExecs = new PrimitiveHookset(MsgReceive.class, a, new PublicOS());
Hookset fAccess = new PrimitiveHookset(FieldAccess.class, new NotCS(a),
                  new OperationSelector(){
                      public boolean accept(Operation op, RClass c){
                        return a.accept(((FieldAccess) op).getTargetType()); });

Hookset useOfA = new CompositeHookset().add(mExecs).add(fAccess);
```

First, since the property of being a subclass of `A` is repeatedly used, we define the class selector `a` matching `A` and its subclasses (as indicated by the `true` parameter). The `mExecs` primitive hookset matches occurrences of the `MsgReceive` operation, provided that they occur in a class accepted by `a` and that they denote public methods (the `PublicOS` is a general-purpose operation selector). Then, the `fAccess` hookset matches occurrences of the `FieldAccess` operation, occurring in a class that is *neither* `A` *nor* a subclass of `A`, and whose target type (the type of the accessed object) is accepted by `a`. Finally, the `useOfA` composite hookset combines public method executions and external field accesses in one entity. This hookset can then be used for instance to log all such events:

```
class Logger { public void log(){ print("access or execution on an A"); } }

BLink log = Links.get(useOfA, new Logger());
log.setControl(Control.BEFORE);
log.setCall("Logger", "log");
log.install();
```

The `log` behavioral link associates occurrences of operations matched by `useOfA` to a newly-created `Logger` object. Reflex supports many means of specifying how the metaobject is obtained (by instantiating a class, querying

a factory, using an existing object, etc). Then, the control of the link is set to `before`, and the call to the metaobject is specialized by saying that the `log` method declared in `Logger` should be called with no parameters.

3.4 Implementation

Reflex is implemented as a Java 5 instrumentation agent operating on bytecode, typically at load time. For each class being loaded, the transformation process consists of *(1)* determining the set of structural links that apply to the class, and *applying* them, and *(2)* determining the set of behavioral links and *installing* them. The reason for this ordering has to do with possible interactions between both kinds of links [40]. During installation of behavioral links, *hooks* are inserted in class definitions at the appropriate places in order to provoke reification at runtime, following the metaobject protocol specified for each link.

3.5 Reflex as an AOP kernel

As a matter of fact, Reflex provides building blocks that facilitate the implementation of different aspect-oriented languages so that it is easier to experiment with new AOP concepts and languages. The flexible model of links for both structural and behavioral reflection can be used as an intermediate target for the implementation of aspect-oriented languages.

This led us to the proposal of *AOP kernels*: versatile substrates for various AOP languages and frameworks, that should make it possible to *compose* aspects written in different AOP languages [31, 41]. At the kernel level, aspect interactions can be detected and resolved. We do not address the details of this layer here, but the interested reader is referred to [40] for a discussion of the concepts and challenges of aspect composition and the various mechanisms supported by Reflex in this regard.

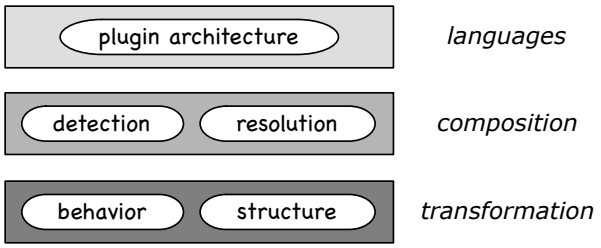

Fig. 3. Architecture of a versatile AOP kernel.

Below this composition layer, a reflection layer implements the intermediate reflective model. This is the layer we have extensively overviewed in this paper.

Above the composition layer, a language layer, structured as a plugin architecture, helps bridge the gap between the aspect models and the intermediate model, as illustrated on Fig. 3. The language plugins part is the most unstable part of Reflex as of today [42], hence we deliberately choose not to enter into details here.

4 Related Systems

There are many systems related to Reflex, may they be tools for structural or behavioral reflection, or tools for AOP in Java. Reflex has however a unique combination of features that is characteristic of the AOP kernel approach: as general as possible reflective features, with advanced customization means, and the direct support for crosscutting changes. In the following, we discuss the salient differences of Reflex compared to existing systems. In-depth discussion of related work in all these areas is out of the scope of this paper, but information can be found in the existing publications on Reflex.

With respect to structural metaobject protocols, Reflex relies on Javassist [22] and hence provides the same facilities for structural reflection. The major difference though is that Reflex follows the design principle of mirror-based reflective APIs [43]: the structural model of Fig. 1 actually consists of *interfaces* rather than direct implementation types as in Javassist. This is crucial for Reflex as a substrate dealing with composition and interaction issues, as it makes it possible to *hide* parts of the real (modified) structure of a class, so that modifications made by one link may or may not be visible to others [40].

Compared to tools for behavioral reflection, like the dynamic proxies of Java, or Kava [23], Reflex offers much better selectivity and expressiveness, within the implementation limits of being a portable library (and not a modified runtime environment, as Iguana/J [44]). In particular, it offers direct support for crosscutting localities, which is typically something missing in most runtime MOPs [28].

Compared to AOP proposals, Reflex is first of all a *framework*, rather than a new language like AspectJ. In this regard, it can be distinguished from other frameworks like AspectWerkz [25] and JAC [26] by its versatility: structural modifications are fully supported, and most importantly, Reflex automatically *detects* interactions between aspects and offers expressive and extensible means for their resolution [40].

Although the core of Reflex is a plain Java framework, the AOP kernel approach in Reflex provides a layer on top of which general-purpose aspect languages like AspectJ, or domain-specific aspect languages like that of SOM [12]

can be defined. Further experiments with concrete syntax support are on-going [42]. A system like XAspects [45] shares the objective of having several aspect languages coexist in a single context, however their approach is based on AspectJ and hence limited to what AspectJ supports. In particular the support for aspect composition is minimal.

5 Conclusion

In this tool presentation paper, we have briefly overviewed the area of metaprogramming, reflection and aspect-oriented programming, and presented the Reflex system for Java. Based on a uniform model of partial reflection, Reflex provides both structural and behavioral facilities, which are well-suited to support aspect-oriented programming. Being a versatile kernel for AOP, Reflex further supports aspect composition as well as the definition of (domain-specific) aspect languages. As a result, Reflex makes it possible to build applications by weaving aspects written in various aspect languages, as well as experiment with old and new AOP languages.

The lower levels of Reflex are stable; both their design and implementation have been validated through quite a number of experiments. Also, part of the design is being exported to Smalltalk (the Geppetto system developed at the University of Bern). We are currently working on extensible concrete syntax with the MetaBorg approach for language embedding and assimilation [46]. We have developed a concrete syntax for Reflex as an alternative to directly using the Reflex API [42]. Also on-going are extensions of Reflex to the realm of distributed computing [16] and context-aware applications [21], which will enhance the applicability of the system.

Acknowledgments. É. Tanter is financed by the Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile. This work has partially been funded by the CoreGRID Network of Excellence and the ITCC Chile-Korea.

References

[1] E. W. Dijkstra, The structure of the THE multiprogramming system, Communications of the ACM 11 (5) (1968) 341–346.

[2] D. Parnas, On the criteria for decomposing systems into modules, Communications of the ACM 15 (12) (1972) 1053–1058.

[3] K. Cheverst, C. Efstratiou, N. Davies, A. Friday, Architectural ideas for the support of adaptive context-aware applications, in: Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality, Bristol, UK, 2000.

[4] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, K. Saikoski, A principled approach to supporting adaptation in distributed mobile environments, in: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000), Limerick, Ireland, 2000, pp. 3–12.

[5] B. C. Smith, Reflection and semantics in a procedural language, Tech. Rep. 272, MIT Laboratory of Computer Science (1982).

[6] P. Maes, Concepts and experiments in computational reflection, in: N. Meyrowitz (Ed.), Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87), ACM Press, Orlando, Florida, USA, 1987, pp. 147–155, ACM SIGPLAN Notices, 22(12).

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (Eds.), Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97), Vol. 1241 of Lecture Notes in Computer Science, Springer-Verlag, Jyväskylä, Finland, 1997, pp. 220–242.

[8] R. E. Filman, T. Elrad, S. Clarke, M. Akşit (Eds.), Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005.

[9] Aspect-oriented software development community and conference, http://aosd.net.

[10] F. Kon, F. Costa, G. Blair, R. H. Campbell, The case for reflective middleware, Communications of the ACM 45 (6) (2002) 33–38.

[11] T. Watanabe, A. Yonezawa, Reflection in an object-oriented concurrent language, in: N. Meyrowitz (Ed.), Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 88), ACM Press, San Diego, California, USA, 1988, pp. 306–315, ACM SIGPLAN Notices, 23(11).

[12] D. Caromel, L. Mateu, É. Tanter, Sequential object monitors, in: M. Odersky (Ed.), Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004), no. 3086 in Lecture Notes in Computer Science, Springer-Verlag, Oslo, Norway, 2004, pp. 316–340.

[13] C. V. Lopes, D: A language framework for distributed programming, Ph.D. thesis, College of Computer Science, Northeastern University (1997).

[14] B. Gowing, V. Cahill, Making meta-object protocols practical for operating systems, in: Proceedings of the 4th International Workshop on Object Orientation in Operating Systems, 1995, pp. 52–55.

[15] L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, D. Suvée, Explicitly distributed AOP using AWED, in: Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), ACM Press, Bonn, Germany, 2006, pp. 51–62.

[16] É. Tanter, R. Toledo, A versatile kernel for distributed aop, in: Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006), Vol. 4025 of Lecture Notes in Computer Science, Springer-Verlag, Bologna, Italy, 2006, pp. 316–331.

[17] J. McAffer, Meta-level architecture support for distributed objects, in: International Workshop on Object-Orientation in Operating Systems (IWOOS 95), 1995.

[18] Y. Yokote, The ApertOS reflective operating system: The concept and its implementation, in: Proceedings of the 7th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 92), ACM Press, Vancouver, British Columbia, Canada, 1992, pp. 414–434, ACM SIGPLAN Notices, 27(10).

[19] R. Rao, Implementational reflection in Silica, in: P. America (Ed.), Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP 91), Vol. 512 of Lecture Notes in Computer Science, Springer-Verlag, Geneva, Switzerland, 1991, pp. 251–266.

[20] L. Capra, W. Emmerich, C. Mascolo, Reflective middleware solutions for context-aware applications, in: A. Yonezawa, S. Matsuoka (Eds.), Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001), Vol. 2192 of Lecture Notes in Computer Science, Springer-Verlag, Kyoto, Japan, 2001, pp. 126–133.

[21] É. Tanter, K. Gybels, M. Denker, A. Bergel, Context-aware aspects, in: Proceedings of the 5th International Symposium on Software Composition (SC 2006) [47], to appear.

[22] S. Chiba, Load-time structural reflection in Java, in: E. Bertino (Ed.), Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000), no. 1850 in Lecture Notes in Computer Science, Springer-Verlag, Sophia Antipolis and Cannes, France, 2000, pp. 313–336.

[23] I. Welch, R. J. Stroud, Kava - using bytecode rewriting to add behavioral reflection to Java, in: Proceedings of USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001), San Antonio, Texas, USA, 2001, pp. 119–130.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: J. L. Knudsen (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), no. 2072 in Lecture Notes in Computer Science, Springer-Verlag, Budapest, Hungary, 2001, pp. 327–353.

[25] Aspectwerkz website, http://aspectwerkz.codehaus.org/ (2002).

[26] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, L. Martelli, JAC: an aspect-oriented distributed dynamic framework, Software Practice and Experience 34 (12) (2004) 1119–1148.

[27] D. Suvee, W. Vanderperren, V. Jonckers, JAsCo: an aspect-oriented approach tailored for component based software development, in: M. Akşit (Ed.), Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), ACM Press, Boston, MA, USA, 2003, pp. 21–29.

[28] É. Tanter, J. Noyé, D. Caromel, P. Cointe, Partial behavioral reflection: Spatial and temporal selection of reification, in: R. Crocker, G. L. Steele, Jr. (Eds.), Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003), ACM Press, Anaheim, CA, USA, 2003, pp. 27–46, ACM SIGPLAN Notices, 38(11).

[29] L. Rodríguez, É. Tanter, J. Noyé, Supporting dynamic crosscutting with partial behavioral reflection: a case study, in: Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), IEEE Computer Society Press, Arica, Chile, 2004.

[30] É. Tanter, From metaobject protocols to versatile kernels for aspect-oriented programming, Ph.D. thesis, University of Nantes and University of Chile (Nov. 2004).

[31] É. Tanter, J. Noyé, A versatile kernel for multi-language AOP, in: R. Glück, M. Lowry (Eds.), Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005), Vol. 3676 of Lecture Notes in Computer Science, Springer-Verlag, Tallinn, Estonia, 2005, pp. 173–188.

[32] É. Tanter, J. Piquer, Managing references upon object migration: Applying separation of concerns, in: Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001), IEEE Computer Society Press, Punta Arenas, Chile, 2001, pp. 264–272.

[33] B. C. Smith, Reflection and semantics in Lisp, in: Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL), 1984, pp. 23–35.

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

[35] R. J. Stroud, Z. Wu, Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press, 1996, Ch. Using Metaobject Protocols to Satisfy Non-Functional Requirements, pp. 31–52.

[36] J. McAffer, Engineering the meta-level, in: G. Kiczales (Ed.), Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96), San Francisco, CA, USA, 1996, pp. 39–61.

[37] G. Kiczales, Towards a new model of abstraction in software engineering, in: Proceedings of the IMSA 92 Workshop on Reflection and Metalevel Architectures, Akinori Yonezawa and Brian C. Smith, editors, 1992.

[38] R. Douence, P. Fradet, M. Südholt, Trace-based aspects, in: Filman et al. [8], pp. 201–217.

[39] K. Lieberherr, I. Silva-Lepe, Adaptive object-oriented programming using graph-based customization, Communications of the ACM 37 (5) (1994) 94–101.

[40] É. Tanter, Aspects of composition in the Reflex AOP kernel, in: Proceedings of the 5th International Symposium on Software Composition (SC 2006) [47], to appear.

[41] É. Tanter, J. Noyé, Motivation and requirements for a versatile AOP kernel, in: 1st European Interactive Workshop on Aspects in Software (EIWAS 2004), Berlin, Germany, 2004.

[42] É. Tanter, An extensible kernel language for AOP, in: Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages, Bonn, Germany, 2006.

[43] G. Bracha, D. Ungar, Mirrors: Design principles for meta-level facilities of object-oriented programming languages, in: OOPSLA 2004 [48], pp. 331–344, ACM SIGPLAN Notices, 39(11).

[44] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behavior, in: B. Magnusson (Ed.), Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002), no. 2374 in Lecture Notes in Computer Science, Springer-Verlag, Málaga, Spain, 2002, pp. 205–230.

[45] M. Shonle, K. Lieberherr, A. Shah, XAspects: An extensible system for domain-specific aspect languages, in: OOPSLA 2003 Domain-Driven Development Track, 2003.

[46] M. Bravenboer, E. Visser, Concrete syntax for objects, in: OOPSLA 2004 [48], ACM SIGPLAN Notices, 39(11).

[47] Proceedings of the 5th International Symposium on Software Composition (SC 2006), Lecture Notes in Computer Science, Springer-Verlag, Vienna, Austria, 2006, to appear.

[48] Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004), ACM Press, Vancouver, British Columbia, Canada, 2004, ACM SIGPLAN Notices, 39(11).