A Flexible Approach to Interactive Runtime Inspection

Éric Tanter[1,2] and Peter Ebraert[1]

[1] University of Chile, DCC/CWR
Avenida Blanco Encalada 2120, Santiago, Chile
[2] Obasco project, École des Mines de Nantes – INRIA
Rue Alfred Kastler, Nantes, France
Eric.Tanter@emn.fr Peter.Ebraert@vub.ac.be

Abstract. Behavioral reflection is well-known approach enabling exhaustive querying of program state (introspection) as well as controlling its execution (intercession). It is hence an adequate foundation for runtime inspection. Partial behavioral reflection aims at making behavioral reflection more applicable by providing high levels of selectivity and configurability. We first outline the main features of partial behavioral reflection and of Reflex—our portable Java implementation. We then sketch how we plan to apply such an approach to provide an interactive environment for runtime inspection, which, in particular, could be used to assist in reflective and aspect-oriented programming.

1 Partial Behavioral Reflection for Java

1.1 Reflection and Behavioral Reflection

Reflection in programming languages is a paradigm that has emerged from the studies of Brian Smith around the foundations of consciousness and self-references, and his work around the application of these concepts to computer science, down to the implementation of a reflective Lisp dialect [1]. These ideas were then applied to various programming paradigms, including object-oriented programming [2] and had a major impact on languages such as CLOS [3] and Smalltalk [4]. In an object-oriented language, reflection is provided through a MetaObject Protocol (MOP), which is an object-oriented "interface" to the language implementation [3].

The basic property of reflection is to support *metacomputations*, that is, computations about computations. This is done by separating metacomputations and *base* computations into two different levels, the *metalevel* and the *base level*. These levels are *causally connected*. This means that, on the one hand, a reflective program running at the base level has access to its representation at the metalevel, and that, on the other hand, a modification of this representation will affect further base computations. Depending on which part of the representation is accessed, the part describing the (static) structure of the program, or the part describing its (dynamic) behavior, reflection is said to be *structural* or *behavioral*.

Another distinction is made between *introspection*, when the representation is simply read, and *intercession*, when the representation is modified.

One of the main strengths of behavioral reflection is to provide the means to achieve a clean separation of concerns including *dynamic* concerns, and hence to offer a modular support for adaptation in software systems [5, 6].

Since behavioral reflection enables exhaustive querying of program state (introspection) as well as controlling its execution (intercession), we believe it is an adequate foundation for runtime inspection. Our focus is reflection-based rather than AOP-based (like in [7]) because we feel behavioral reflection is the essence of generic runtime AOP approaches. The major point of AOP, to us, lies in the definition of Aspect-Specific Languages (ASLs) on top of generic (possibly reflective) infrastructures (see for instance the work of [8]). We therefore start by presenting partial behavioral reflection and Reflex, before sketching how we plan to apply our approach to create an interactive runtime inspection environment. We are particularly interested in applying such an environment to help programmers develop/debug their "metalevels".

1.2 Partial Behavioral Reflection

Bringing behavioral reflection into a language such as Java raises new challenges, compared to languages like Smalltalk. Indeed, Java only provides very limited reflective facilities[1] and, in order to reduce runtime errors and take into account security requirements, is much more static in nature. As a result, the overhead due to the additional layer necessary to get behavioral reflection increases significantly. Selecting where and when to apply reflection becomes mandatory. This is called *partial reflection* [9]. Partial reflection makes it possible to balance the effects of compilation–which *embeds* a set of assumptions (a specialization), and reflection–which *retracts* some of these assumptions (a generalization).

The starting point of our proposal is the following: the metalevel is structured in terms of *metaobjects* reasoning and acting upon *reifications* of the base level computation described in terms of operations (e.g., message send/receive, field access, cast, object creation, serialization, etc.). Reifications are object representations of *base-level operation occurrences* (also known as *execution points*) which metaobjects can manipulate (Fig. 1).

We are developping an extensive approach to partial behavioral reflection that relies on avoiding useless reifications. To this end, high flexibility in specifying reflective needs is required. Our approach is based on spatial and temporal selection of reification [10]. Spatial selection refers to the possibility of precisely specifying *what*, in an application, should be reified (which operation occurrences, in which objects/classes). Temporal selection refers to the possibility of specifying *when*, during an application's lifetime, particular reifications effectively occur. The related ideas and techniques would be applicable to a wide range of object-oriented languages, although we focus on the Java language.

[1] The standard Java reflection API mainly provides structural introspection. Since the JDK 1.3, the Dynamic Proxies provide a restricted kind of behavioral intercession based on interception objects.

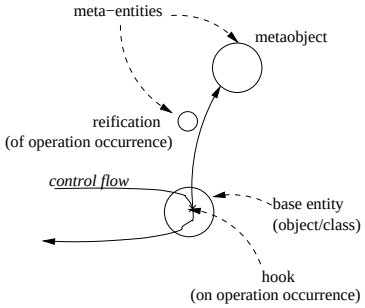

Fig. 1. Behavioral reflection with metaobjects

1.3 Reflex

Reflex is an open runtime metaobject protocol (MOP) for Java supporting partial behavioral reflection with a high level of configurability, selectivity and flexibility.

Hooks and Hooksets. Reflex is based on the notions of *hooks* and *hooksets*, which are similar to the AOP notions of join points and pointcuts. A hook is a piece of code inserted to reify a particular operation occurrence, while a hookset is a set of related *hooked* execution points. The Reflex core framework in itself does not support any operation, but rather, Reflex reification capabilities are extended by providing reification components for desired operations. The standard Reflex library includes a comprehensive set of such components. Hooksets can gather execution points located in distinct entities, thus making it a convenient way to handle crosscutting concerns.

Reflex offers an expressive framework for defining hooksets declaratively. Users can define hooksets and associate metaobjects to them either statically, using configuration files, or dynamically, through a runtime API. The flexibility of the framework allows for specification of various attributes regarding hooksets and metaobjects, in particular:

- several hooksets may intervene within a given object/class
- several hooksets acting at the same execution point may be composed
- metaobjects can be object-, class- or hookset-specific (i.e., customizable *scope*).
- metaobjects may intervene before and/or after base operation occurrences, or may be more powerful and completely replace the semantics of base operation occurrences (i.e., customizable metaobject *control*). Well-defined operation- and control-specific metaobject interfaces are provided.
- hooksets can have a *dynamic activation condition* attached to them, to control their activation/deactivation. Such activation condition can depend on

both base-level and metalevel data. This makes it possible to reason on control flow (similar to `cflow` in AspectJ [11]) or sequences of events (like in EAOP [12]) to determine dynamically if a given hookset should be active or not.

Figure 2 illustrates our model of hooksets and metaobjects, which is similar to event-condition-action models. In our case, event sources (hooks) can be activated/deactivated as needed, activation condition are dynamically evaluated and can be changed at any time, and actions (taken by metaobjects) can be adapted and composed dynamically too.

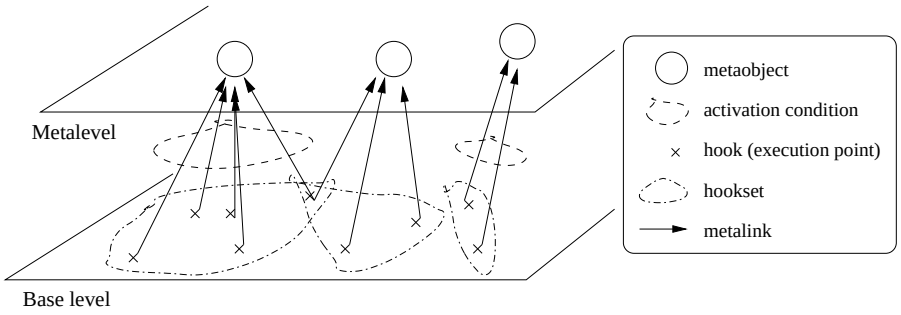

Fig. 2. The model of hooksets and metaobjects.
In this example, three hooksets (two of which are activatable) are represented with their associated metaobjects.

Implementation approach. Our implementation approach is based on the idea of maximizing flexibility and dynamicity while preserving portability, which we consider a major advantage of Java. Hence we do not extend or modify the Java Virtual Machine but instead Reflex is a standard Java library using load-time instrumentation for inserting hooks. The Java Platform Debugger Architecture (JPDA) [13] also represents an attractive medium to implement runtime inspection but, at present, it seems far from being a state-of-the-art environment, in particular due to its restricted expressivity: few language events are supported (method entry/exit, field access and exception) compared to the more comprehensive expressivity that can be obtained using bytecode transformation (e.g., cast, object creation, serialization).

Our approach has the disadvantage that once a class is loaded, new hooks cannot be inserted into it[2]. However, as mentionned above, we provide *activatable* hooks, which have an almost negligible cost when deactivated (preliminary

[2] Although this limitation could be overcome using hot swapping of classes.

benchmarks can be found in [10]). It is hence possible to systematically install deactivated hooks in places of potential interest, knowing that these hooks can be activated later on. Hence, dynamicity is *constrained*, but *adjustable*, and portability is preserved.

2 Towards an Interactive Environment for Runtime Inspection

2.1 Requirements for runtime inspection

An interactive environment for runtime inspection has several requirements related to software visualization issues [14]. Although the core of this position paper is not about visualization itself (these aspects would require more research), we believe at least two important issues deserve early consideration: visual load, and synchronization.

As soon as we are interested in realistic applications, a tough issue in visualization is the control of the *visual load*, that is, the possibility to ensure that not too much information is displayed at a given time, so that the user cognitive charge is not excessive and the user can avoid getting lost. A user should be able to select what exactly is of interest to him and possibly be given the chance to adjust the visualization layout. Another important requirement deals with the *synchronization* between the executing application and the inspection environment. Our approach is based on a synchronous *means of control* [7]. The idea here is to provide the user with a feeling of direct interaction with the running application, offering the possibility to suspend execution, to adjust settings or interact with the application, before resuming execution. To sum up, it seems fundamental to provide a very fine-grained control over what is inspected, along with a high-level of interactivity.

2.2 Approach to Application Inspection

According to the above-mentioned requirements, we are convinced that our approach to partial behavioral reflection is particularly well-suited to provide an interactive environment for runtime inspection. In order to back up that statement, we are developping a first monitoring tool that uses Reflex for allowing runtime inspection.

First of all, to specify which parts of a base application a user wants to monitor, Reflex configuration files are used to define *hooksets*. In addition to this static configuration, the runtime API of Reflex can be used to control hookset activation. This makes it possible for the user to precisely control, down to the finest granularity level, which parts of the system and which particular execution points will be observed/manipulated and when. For instance, a user can specify that monitoring a given part of the system becomes interesting whenever some dynamically-evaluated activation condition holds.

Visually, the user gets confronted with a multi-window system where a small control window is attached to every hookset (a *GUI metaobject* is associated

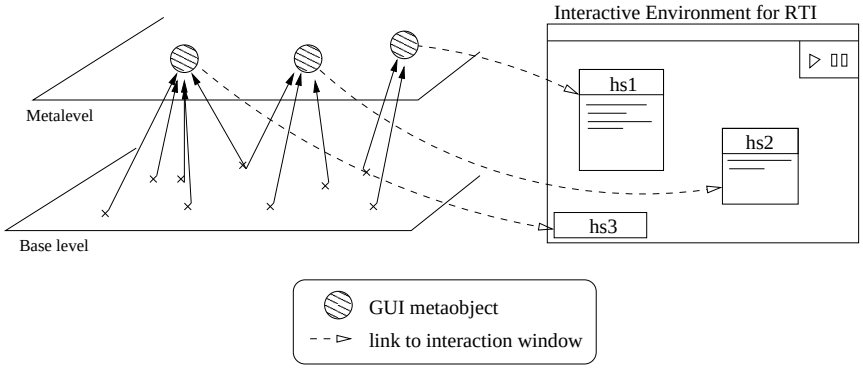

Fig. 3. Schema of the interactive environment.

to each hookset). This permits the visualization layout to be customized (e.g. windows can be minimized, resized, closed and moved). At present, a control window simply offers terminal-like output and basic interaction features:

- inspection of reified operations with a notion of timeline: to indeed introspect the base application execution,
- custom speed execution of the base application: to control the amount of information presented to the user per time unit, and
- fine-grained control over hookset activation conditions: to limit the set of displayed events.

For each hookset, a window is created, responsible for the monitoring of its operations. Depending on the hookset scope, different types of windows are used:

- For a hookset-scope hookset, the window is a simple hookset-scope window that monitors all operations affected by the hookset.
- For a class-scope hookset, the hookset window contains a class-scope window for each of the classes affected by that hookset. Each class window monitors the operations related to the considered hookset occurring within its instances.
- For an object-scope hookset, the hookset window also contains a class-scope window for each of the classes affected by that hookset. But in this case, each class window contains an object-scope window monitoring only the operations related to the considered hookset occurring within a particular instance.

Closing a window on a certain level closes all its nested windows and stops the monitoring of that hookset, class or object. This allows visual load to be limited as the user can really select what he wants to monitor. Figure 3 shows how the interface looks like. Obviously, further versions could illustrate program dynamics in a more elaborated fashion.

Synchronization between the user and the running application is made at each entry to the metalevel. Whenever a reification occurs, the interactive environment can synchronize with the running application. For instance, if the user asks to suspend the program execution, this suspension will take effect upon the next entrance to the metalevel. Customizing the base application execution speed is also managed this way.

2.3 Perspectives for Concerns Inspection

Reflex is a general tool for supporting separation of concerns (SOC) through behavioral reflection. Indeed, it can serve as a generic platform for aspect-oriented development as argued in [10]. In such a case, an application runs at the base level while crosscutting and/or non-functional concerns are implemented modularly as metalevel entities (*SOC metaobjects*).

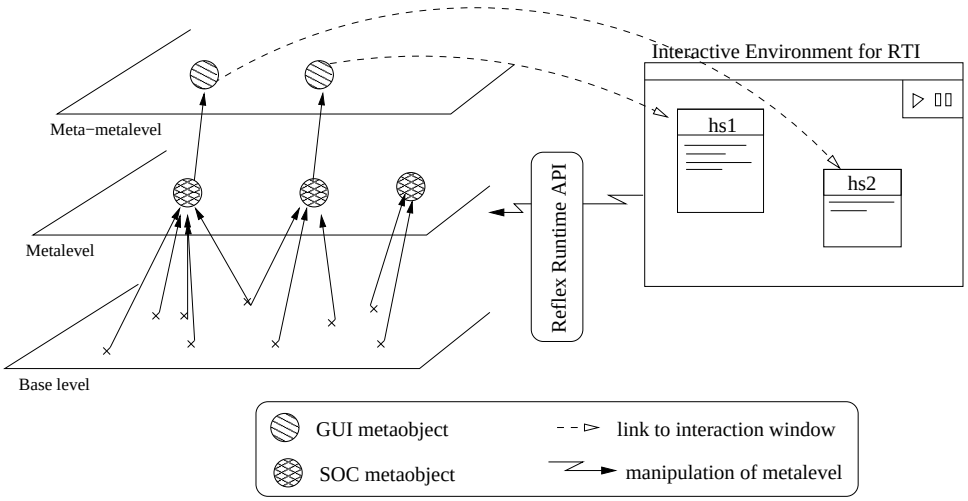

Fig. 4. Applying the interactive environment to inspect a *concern* metalevel.

In order to help in the prototyping, development, and debugging of such a *concern* metalevel, our approach to runtime inspection could provide a valuable support. In such a scenario, the inspection environment actually runs at the meta-metalevel, allowing for the manipulation of the metalevel (Fig 4). The interaction with the runtime API of Reflex would then be much more powerful than in standard application inspection, since it would not simply include hookset activation and basic synchronization features, but would also provide the means to dynamically change the bindings between base execution points and SOC metaobjects.

Note that plugging a metalevel on top of an existing metalevel does not raise any problem since metaobjects are implemented with standard Java classes, and are thus also subject to selective reification.

3 Conclusion

In this paper we first briefly presented partial behavioral reflection and its portable implementation in Java, Reflex. We have highlighted how hooksets provide a customizable means to conceptually group execution points of interest, and manipulate, possibly dynamically, the metaobjects associated with them. Hooksets may also be subject to dynamically-evaluated activation conditions. We have then explained why we consider that this approach is a convenient way to provide an interactive environment for runtime inspection and we have sketched how we have started to explore in this direction. Finally, the recursivity of the model of behavioral reflection makes it possible to apply such a graphical tool to inspect a metalevel layer implementing crosscutting and/or non-functional concerns. This would certainly prove highly useful in assisting the prototyping, development and debugging of applications making use of runtime SOC.

Acknowledgements

This work is partially funded by Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

References

[1] Smith, B.C.: Reflection and Semantics in Lisp. In: Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages. (1984) 23–35
[2] Maes, P.: Computional Reflection. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium (1987)
[3] Kiczales, G., Des Rivires, J., Bobrow, D.: The Art of the Meta-Object Protocol. MIT Press (1991)
[4] Rivard, F.: Smalltalk: a Reflective Language. In: Reflection'96. (1996)
[5] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran, H., Parlavantzas, N., Saikoski, K.: A Principled Approach to Supporting Adaptation in Distributed Mobile Environments. In: Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'2000). (2000)
[6] Redmond, B., Cahill, V.: Supporting Unanticipated Dynamic Adaptation of Application Behavior. In: Proceedings of ECOOP 2002. Volume 2374 of Lecture Notes in Computer Science., Málaga, Spain, Springer-Verlag (2002) 205–230
[7] Mehner, K., Rashid, A.: Towards a Standard Interface for Runtime Inspection in AOP Environments. In: Workshop on Tools for Aspect-Oriented Software Development at OOPSLA 2002. (2002)
[8] Brichau, J., Mens, K., De Volder, K.: Building Composable Aspect-specific Languages. [15]

[9] Ibrahim, M.H.: Report of the Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming. In: OOPSLA/ECOOP'90, Ottawa, Canada (1990)
[10] Tanter, E., Noyé, J., Caromel, D., Cointe, P.: Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. To Appear at OOPSLA 2003 (2003)
[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. Proceedings of ECOOP 2001 (2001)
[12] Douence, R., Fradet, P., Südholt, M.: A Framework for the Detection and Resolution of Aspect Interactions. [15]
[13] Sun Microsystems, I.: The Java Platform Debugging Architecture. `http://java.sun.com/products/jpda` (2001)
[14] Stasko, J., Domingue, J., M.H., B., Price, B.e.: Software Visualization. The MIT Press (1998)
[15] Batory, D., Consel, C., Taha, W., eds. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002). Volume 2487 of Lecture Notes in Computer Science., Pittsburgh, PA, USA, Springer-Verlag (2002)