# Higher Abstractions for Dynamic Analysis[*]

Marcus Denker, Orla Greevy
Software Composition Group
University of Berne
Switzerland

Michele Lanza
Faculty of Informatics
University of Lugano
Switzerland

## ABSTRACT

The developers of tools for dynamic analysis are faced with choosing from the many approaches to gathering runtime data. Typically, dynamic analysis involves instrumenting the program under investigation to record its runtime behavior. Current approaches for byte-code based systems like Java and Smalltalk rely often on inserting byte-code into the program under analysis. However, detailed knowledge of the target programming language or virtual machine is required to implement dynamic analysis tools. Obtaining and exploiting this knowledge to build better analysis tools is cumbersome and often distracts the tool builder from the actual goal, which is the analysis of the runtime behavior of a system.

In this paper, we argue that we need to adopt a higher level view of a software system when considering the task of abstracting runtime information. We focus on object-oriented virtual machine based languages. We want to be able to deal with the runtime system as a collection of reified first-class entities. We propose to achieve this by introducing a layer of abstraction, *i.e.*, a behavioral middle layer. This has the advantage that the task of collecting dynamic information is not concerned with low level details of a specific language or virtual machine. The positive effect of such a behavioral middle layer is twofold: on the one hand it provides us with a standard API for all dynamic analysis based tools to use, on the other hand it allows the tool developer to abstract from the actual implementation technique.

## Keywords

Dynamic Analysis, Behavioral Reflection, Meta Programming, Tracing

## 1. INTRODUCTION

In recent years there has been a revival of interest in dynamic analysis [16]. System analysis of runtime behavior is vital for performance analysis to detect hotspots of activity and bottlenecks of execution or memory allocation problems such as unnecessary object retention. In a reverse engineering context, dynamic analysis is used is to extract high-level views about the behavior of low-level components to facilitate the comprehension of the system [15, 17, 31]. The focus of analysis determines the relevance and level of detail of the information captured during dynamic analysis. In the field of reverse engineering, there is no consensus on the type or level of granularity of runtime information that is necessary for a particular analysis. An inherent requirement of such tools is that they be easily extensible as the requirements and the research focus evolves.

Dynamic analysis yields precise information about the runtime behavior of systems [2]. However, the task of writing tools to abstract runtime data is not trivial. Developers of tools are faced with many options as there are numerous techniques that address the task of collecting runtime data. The approach to tool development and the abstraction of dynamic data is therefore not standardized. Each individual tool adopts a specific technique and focuses on low-level details of the chosen technique to achieve its goals.

This leads to recurrent problems of all approaches and techniques:

- all clients need to re-implement large parts of the code responsible for gathering the runtime data, and
- the abstraction level is too low in the sense that clients need to know too much about the implementation side.

In this paper we propose the introduction of an abstraction layer based on *behavioral reflection* to facilitate the design and development of tools for dynamic analysis. We introduce our reflection framework and identify its strengths and shortcomings.

**Structure of the paper.** In the next section we identify some typical applications of dynamic analysis. In this context we outline the state of the art in gathering dynamic data at runtime. Section 3 then shows problems and shortcomings associated with the current approaches. In Section 4 we give an overview of reflection frameworks. In Section 5 we introduce our reflection framework and identify how it can be used to solve the problems shown, and we identify what is missing from our implementation with the intention of initiating a discussion and obtaining feedback. Section 7 outlines our conclusions and future work.

## 2. DYNAMIC ANALYSIS TECHNIQUES

Dynamic analysis typically involves instrumenting the program under investigation to examine or record certain aspects of its runtime behavior. Code instrumentation is a mechanism that allows insertion of code at runtime to monitor and track the runtime behavior of a system. In this section we review the techniques currently available for abstracting the runtime behavior of a system. The underlying concepts behind dynamic analysis tools are currently limited

---

[*] Proceedings of the 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006), pp. 32-38

to using these techniques for extracting dynamic information [17].

The granularity and amount of behavioral data extracted during the execution of a system varies depending on the specific focus of a particular analysis tool. Dynamic analysis implies vast amounts of data. A simple execution of a system's functionality can result in a large number of events [10]. Typically, dynamic analysis tools employ filtering and compression techniques to limit the amount of dynamic data collected depending on a specific focus of the analysis. For example, if the goal of the analysis is feature location [11], this requires that a relationship between the external functionalities of a system and the parts of the code that implement this functionality is established. Therefore, it is usually sufficient to extract trace events representing the method calls performed during the execution of a specific functionality [1, 15]. An example of trace representation is given by Richner and Ducasse [27]. Each line records the class of the sender, the identity of the sender, the class of the receiver, the identity of the receiver and the method invoked. The order of the calls is maintained.

Analysis techniques that focus on monitoring the state of objects at runtime require a more detailed analysis to extract information about variable access. This level of granularity is required if the focus of the analysis is to infer program invariants [12]. Performance analysis tools generally focus on object creation and deletion and the correct memory allocation details. Thus the requirements of dynamic analysis tools vary depending on their specific focus. This is a drawback, because the analysis goals should not restrict the type of information to be collected. We want to extract as much dynamic data as possible and then depending on the requirements of a particular analysis, extract and use a appropriate subset of the dynamic data.

There are various approaches and language-specific frameworks that support the extraction of dynamic information. We describe the details of the underlying mechanisms used by dynamic analysis tools in the following paragraphs.

**Source code modification.** One way to control message passing is to instrument the code via source code modification and recompilation. The main drawback of this technique is that all controlled methods have to be reparsed and recompiled. Furthermore, another recompilation is needed to reinstall the original methods.

**Bytecode modification.** Another way to control message passing is to directly insert byte-code into the byte-code of the compiled methods. The introduced byte-code controls the message passing. However, this technique relies heavily on profound knowledge of the bytecode instructions used by the virtual machines. Another potential danger of this technique is that these codes are not standardized and can change.

**Instrumenting the Virtual Machine.** A technique for collecting runtime information is instrumenting the runtime environment in which the system runs. For example, the Java Virtual Machine can be instrumented to generate events of interest. The advantage of this technique is that it does not require modification of the source code.

The Java Virtual Machine Profiling interface (JVMPI) [20] provides a set of hooks to the JVM to signal interesting events, such as thread starts or object allocations. JVMTI [21] is the sucessor to JVMPI and provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine. It provides additional facilities for bytecode instrumentation. Profilers based on JVMPI or JVMTI interfaces implement profiling agents to intercept various events, such as method invocations. Unfortunately these profiling agents have to be written in platform native code, contradicting the Java motto of "write once run anywhere". Binder developed Komorium, a novel sampling profiler written purely in Java that directly instruments the bytecode of Java programs [4]. Another pure Java solution is the Java Interactive Profiler (JIP) is based on JVMTI and provides control to turn on and off profiling at runtime (see `http://jiprof.sourceforge.net/`).

**Method Wrappers.** Brant *et al*, describe a code instrumenting technique for Smalltalk based on method wrappers [5]. Wrappers are mechanisms for introducing new behavior that is executed before and/or after, and perhaps instead of, an existing method. Rather than changing method lookup, they modify the method objects that the processes return. Wrappers are easy to build for Smalltalk as it was designed with reflective facilities that allow programmers to intervene in the lookup process, while the same is not true for Java which only supports introspection.

**Debuggers.** It is possible to run a system under the control of the debugger. In this case, breakpoints are set at locations of interest (*e.g.*, entry and exit of a method). This technique has the advantage of not modifiying the source code and the environment. However it slows down the execution of a system considerably, and the instrumentation itself can be cumbersome. This can be done on the source level before compilations, or on bytecode at load time (Java) or runtime (Smalltalk). The abstraction layers we deal with are either those of the program text or those of bytecode.

**Logging Services.** Logging can be used to track the state of a running system at various points in time. A good framework for doing this with Java is provided by log4j (see `http://logging.apache.org/`). The drawback is again that this implies modifying the source code.

## 3. CHALLENGES

As we have seen, there are multiple implementation techniques for gathering runtime data. The key problem is that every new client application has to re-implement large parts of the runtime data gathering code depending on the language and runtime environment. Furthermore, the abstraction is too low level, resulting in clients that are concerned with manipulating too many implementation details.

In the following sections we elaborate on the main problems of the current approaches.

### 3.1 Instrumentation re-implemented

Most projects that require to access runtime data typically re-implement the data gathering mechanism. Instrumentation code is inserted at all places of interest in the code (*e.g.*, at message sends). In the case of bytecode manipulation techniques, the actual modification of the bytecode is achieved using libraries (*e.g.*, Javassist [8, 7] or Bytesurgeon [9]). Bytecode manipulation provides a very low level of abstraction and requires detailed knowledge of the bytecode of the programming language. Moreover, it is prone to language evolution, *i.e.*, the specification of the VM may change.

The positive effect of the low level approach is of course that we build a very specific implementation, tailored exactly towards the information needed for a specific task. The drawback is that the instrumentation logic is tightly coupled with the application that requires the dynamic data, thus in most cases we will have to re-implement the instrumentation logic for each new task. Figure 1 shows an example: We have two projects that are interested in gathering runtime data (Tracer1 and Tracer2). Although both run on a standard virtual machine, both independently implement the code for bytecode instrumentation. We have seen this happen often in our research, for example both the trace debugger Unstuck [19] and a test coverage tool both utilized the same byte-code manipulation library (ByteSurgeon), but they did not share any instrumentation code.

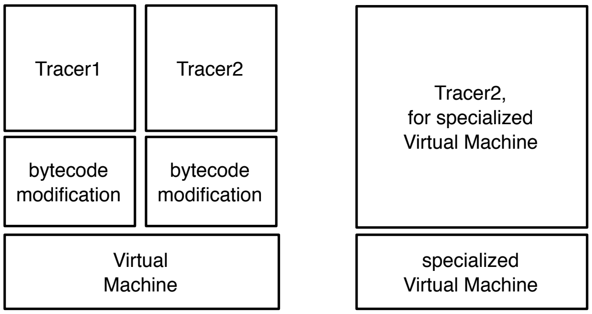

**Figure 1: Trace tool today**

## 3.2 Implementation Specific Designs

Implementors of tools that gather runtime data are faced with the decision on which implementation technique to adopt. Subsequently, they design a tool based on specific knowledge of the target language and runtime environment. The resulting tool inevitably is forced to encode implementation issues into the design of the tool. Thus, the result is a tool that is tightly coupled with a particular runtime environment.

This approach has obvious drawbacks. It is very difficult to change the adopted implementation technique: *e.g.*, bytecode manipulation is portable, but a specialized virtual machine might be faster. When the implementation technique is closely tied to a particular virtual machine, we are bound to this one implementation, we cannot switch to a byte-code based implementation on a case-to-case basis. Figure 1 shows that for a special virtual machine, we need to re-implement our system.

# 4. BEHAVIORAL REFLECTION

Systems like Smalltalk and CLOS model their own structures (classes, methods) as first class objects.

The term *introspection* defines the ability to query the system about this information, whereas we talk about *intercession* when changes to these structures directly change the structure of the system itself. Systems that provide both are called *reflective*.

Structural reflection is concerned with reification of the program and its abstract types. *Behavioral reflection*, on the other hand, is concerned with the ability of the language to provide complete reification of its own semantics and implementation as well as complete reification of the data and implementation of the runtime system.

Popular object oriented languages provide different degrees of introspection or reflective capabilities. Smalltalk is, to some extent, a reflective system [13, 3]: Classes and methods are objects, we can change those objects to change the structure of the system. Java and .NET on the other hand, have only introspective features (*i.e.*, allows for querying an object for its class, a class for its methods and constructors, query the details of those methods and constructors, and tell those methods to execute), and partial intercession (intercession is limited to method invocation and attribute manipulation) [6].

Languages that facilitate the description of methods as first class objects inherently support dynamic analysis. The method wrappers technique exploits the first class nature of methods in Smalltalk for providing a way to intercept method execution [5]. Examples of dynamic analysis tools built on the method wrapper technique are Greevy and Ducasse's TraceScraper tool for feature analysis [15] and John Brant's Interaction Diagram and Coverage Tools [5]. However method execution is just an aspect of runtime information. For a complete dynamic analysis we need to focus on other runtime events such as *e.g.*, *message sends between object instances* or *instance variable access*. Thus, we recognize the need to define a reflective meta representation that describes all behavioral aspects of systems. We want a system that can reify those events on demand, providing a system with full behavioral reflection.

In both Java and Smalltalk, the reflection mechanisms provided are concerned mostly with structure. They do not provide an easy way to change the semantics of the runtime model: Message sends, instance variable access are not modeled with objects. A true behavioral reflective system models behavior in a way that it is first class and changes are possible: *e.g.*, we are able to define our own version of what a message send is.

Looking back into the history of object oriented systems, we can find that there has been extensive research on behavioral reflective systems, *e.g.*, the work done around Meta Object Protocols [22] for CLOS. The meta object protocol provides all operations (*e.g.*, method activation or variable access) to be reified and re-defined.

In systems like Java and Smalltalk, behavioral reflection can be realised via special virtual machines or bytecode manipulation, with the latter being portable. Examples for the virtual machine approach are Iguana/J[26], Metaxa [14], or Guarana [25]. The prime example for a bytecode modification based meta object protocol is Reflex [30]. Reflex provides a model for behavioral reflection that allows for a very fine grained selection of when and what to reify.

# 5. THE BEHAVIORAL FRAMEWORK

The drawbacks we have identified with current approaches lead us to suggest that the solution would be to introduce an additional layer of abstraction to our system, which we refer to as a *behavioral framework*.

We now analyze how a behavioral reflection framework could be used to tackle and solve the problems of previous approaches to gathering runtime information.

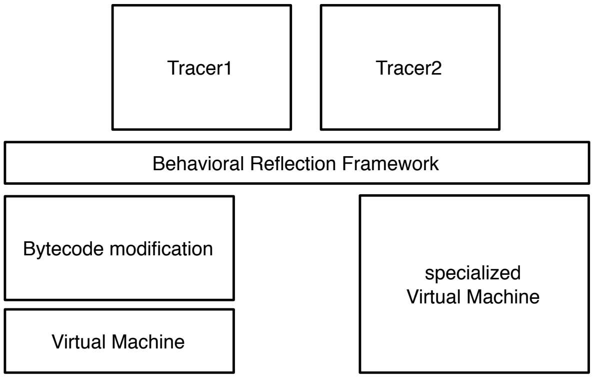

**Figure 2: A common abstraction layer**

## 5.1 A Shared API

With the existence of a behavioral layer, all tools could use it to hook into runtime events. The individual tools are no longer concerned with a specific code insertion implementation. Instead, they just leverage the abstractions provided by the behavioral layer framework.

In Figure 2 we see again our two tools that are interested in dynamic information. Now both tools just use the behavioral layer, thus they do not need to implement the byte-code modification code themselves, but share it.

## 5.2 A Pluggable Implementation

Another important requirement of an abstraction layer is to provide a high degree of flexibility, but at the same time retain the same interface for clients. The proposed behavioral framework should make it possible to have a pluggable implementation (the backend): it can be realized via editing byte-code, a changed virtual machine or other means.

Figure 2 shows how we now can use both programs on the modified virtual machine, without having to implement the logic ourselves: All tools using the abstraction layer will work on both the standard virtual machine and any specialized virtual machine that the abstraction layer supports.

## 5.3 Requirements

In the following we identify a list of requirements for a behavioral framework to tackle the challenges we identified previously.

**Runtime installation:** We need to introduce behavior dynamically at runtime. When we are done with the analysis, it should be possible to revert to the original state of the system.

**Unanticipated use:** The behavioral change should be possible at any time in the deployed system, without the need to prepare the system in any way at startup.

**Fine-Grained Selection:** The operation occurrences we are interested in are different depending on what we analyze. We want to be able to select the entities up to the level of the single occurrence in the code.

**Implementation Hiding:** From a dynamic analysis perspective, we are not interested in the underlying mechanisms of obtaining runtime information. The fundamental goal of a behavioral layer is to allow us to abstract from the details of a specific implementation technique (*e.g.*, VM change, byte-code extension, byte-code modification) used to extract behavioral information from an application at runtime.

**Performance:** To make the framework usable for analyzing real work applications, we need a framework with low overhead. The best case would be a system where we pay exactly the same overhead as if we were to annotate the code with profiling calls by hand.

## 5.4 Implementation

We have realized a framework for partial behavioral reflection for Squeak (a dialect of Smalltalk) called Geppetto[28]. Geppetto uses the runtime byte-code transformation framework ByteSurgeon[9] and follows the model of partial behavioral reflection as pioneered by Reflex[30]. Unlike Reflex, which is constrained by the underlying model of the Java language, our Geppetto implementation can be used completely unanticipated: code does not need to be prepared at load or compile time, reflection can be enabled at runtime and completely retracted when not needed.

Geppetto allows for reifying message sending, method execution and variable access (read and write) for both instance variables and temporary variables. Selection is very fine-grained: per package, class, object, method, operation and operation occurrence. Geppetto can be used in any Squeak program, without the need to adapt it at load or start time. Installation happens transparently at runtime.

Geppetto uses ByteSurgeon to insert small peaces of code, so called *hooks* into the bytecode where a selected operation (e.g. message send) occurs. Figure 3 shows the model in detail. Hooks are grouped to *hooksets*, which are bound to a *metaobject* by a *link*. The *link* defines the protocol between the base and the meta layer. Links can be enabled or disabled based on an *activation condition*.

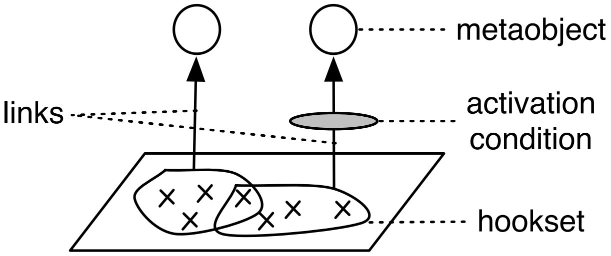

**Figure 3: Hooksets, Links and Metaobjects in Geppetto**

For a complete description of the Gepetto behavioral reflection framework, see [28].

## 5.5 Usage

The behavioral reflection framework provides a general API: the reification of runtime events triggers calls to meta objects, which are instances of normal classes. The tool developer thus is free to use the framework as needed by specifying which concepts to reify and which information to pass on to the meta object. The framework does not provide a model of the data obtained (*e.g.*, a trace), instead it provides *a model for obtaining data*. It can be either stored for later use as a trace or processed and reacted on at runtime. The latter has lately become an active topic of research with systems like PQL [24].

# 6. DISCUSSION

We now analyze our behavioral framework with respect to the requirements defined in the preceding section and define future work. Then we briefly discuss the relationship to aspect oriented programming and the usefulness of providing scoping abstractions as part of the framework.

## 6.1 Next Steps

The implementation as described in section 5.4, already fulfills some of the requirements stated: It can install (and retract) behavioral changes at runtime, provides fine-grained spatial and temporal selection by implementing the Reflex model [30] and supports unanticipated use.

Two requirements are not yet fulfilled:

1. Geppetto needs to be extended to support pluggable backends. We are working on providing a backend based on annotated abstract syntax trees.

2. We need to verify the real world usability: first benchmarks show good performance characteristics, but Geppetto needs to be validated with real world usage. We plan to move the tools and experiments done that currently use ByteSurgeon to use Geppetto instead.

## 6.2 Aspects

This paper presents the solution from the perspective of behavioral reflection. Another point of view can be that of Aspect Oriented Programming. The proposed abstraction layer could use, as a backend, an existing dynamic aspects implementation. In this case, the aspect framework would be used as a high-level replacement for byte-code manipulation.

Another possibility would be to formulate the middle layer in terms of a dynamic aspect framework instead of meta objects. The problem here is that most aspect systems (*e.g.*, AspectJ [23]) are static: weaving happens at compile or load time. Pure runtime Aspects are not yet very common and those that exist are based themselves in some cases on behavioral reflection facilities, for example AspectS[18] and aspect systems based on Reflex[29].

## 6.3 Scope Abstractions

Modern implementations like Reflex provide very fine-grained spatial and temporal selection of reification. Here we can select *what* and *where*, in a temporal and spacial way.

This means we can scope the reification towards collections of classes (like modules and packages) or single instances, a single methods of a class, or even to one certain occurrence of a behavioral event. Temporal selection means that we can switch reifications on and off at will, thus we can make the gathering of runtime data be controlled by runtime events.

Another idea of scoping is that of scoping-towards-the-client: We might be interested in events generated only if our package under test is called from a certain other package. This can be useful to limit the amount of unnecessary data when *e.g.*, analysing system classes like Smalltalks collections.

# 7. CONCLUSION

In this paper we addressed a fundamental problem that faces the developers of tools that exploit runtime information of an application. We propose an new approach to designing dynamic analysis tools for virtual machine based languages that interact with a layer of abstraction, namely a behavioral layer. The behavioral layer should provide a framework for tool developers that encapsulate typical object oriented language constructs at runtime such as object instantiation, message sends and instance variable access. Thus the developer has access to reified first class entities of runtime behavior and focuses on these high level abstractions when designing a specific tool. The main advantage of this layer of abstraction is that the resulting tool should easily portable to use with other virtual machines as the reified entities are independent of the underlying implementation details and byte-codes. Moreover the developer is not concerned with low level details that are specific to a particular virtual machine.

In this paper we provided a short overview of the available technologies and approaches to extract runtime data. We identified problems inherent to these approaches. This motivates our argument that there is a need to introduce a layer of abstraction between low level implementation details and the tools analysing the data.

To better understand the underlying motivation of a behavioral layer we provided a short overview of some of the applications of dynamic analysis. In the field of program comprehension and reverse engineering dynamic analysis approaches are becoming more prevalent. However there is no standard approach to extracting runtime data nor is it clear which type of runtime information to extract. Therefore such tools need to be extensible, as requirements change.

We identified a list of requirements for a behavioral layer. We describe our current implementation of a behavioral layer and illustrate how it can be used to address the problems. We show how we simplify the task of implementing dynamic analysis tools.

**Acknowledgments.** We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008)) and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997).

# 8. REFERENCES

[1] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 357–366,

Los Alamitos CA, September 2005. IEEE Computer Society Press.

[2] Thomas Ball. The concept of dynamic analysis. In *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, number 1687 in LNCS, pages 216–234, Heidelberg, sep 1999. Springer Verlag.

[3] Alexandre Bergel and Marcus Denker. Prototyping languages, related constructs and tools with Squeak. In *In Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[4] Walter Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems (APLAS-2005)*, volume 3780 of *LNCS*, pages 178–194, Tsukuba, Japan, nov 2005.

[5] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998. method wrappers.

[6] Walter Cazzola. Smartreflection: Efficient introspection in java. *Journal of Object Technology*, 3(11), August 2004.

[7] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of GPCE'03*, volume 2830 of *LNCS*, pages 364–376, 2003.

[8] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336, 2000.

[9] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[10] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[11] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of ICSE '99*, May 1999.

[13] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, October 1989.

[14] Michael Golm and Jürgen Kleinöder. Jumping to the meta level: Behavioral reflection can be fast and flexible. In *Reflection*, pages 22–39, 1999.

[15] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society Press.

[16] A. Hamou-Lhadj. The concept of trace summarization. In *Proceedings of PCODA 2005 (1st International Workshop on Program Comprehension through Dynamic Analysis)*. IEEE Computer Society Press, 2005.

[17] A. Hamou-Lhadj and T. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*, pages 42–55, Indianapolis IN, 2004. IBM Press.

[18] Robert Hirschfeld. AspectS – aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[19] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, 2006.

[20] Sun microsystems, inc. jvm profiler interface (jvmpi).

[21] Sun microsystems, inc. jvm tool interface (jvmti).

[22] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[24] Mickael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 363–385, New York, NY, USA, 2005. ACM Press.

[25] A. Olivia and L. E. Buzato. The design and implementation of Guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 203–216, San Diego, California, USA, May 1999.

[26] 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.

[27] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2002)*, page 34, Los Alamitos CA, October 2002. IEEE Computer Society Press.

[28] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection. In *Proceedings of ISC 2006 (International Smalltalk Conference), LNCS, to appear*, 2006.

[29] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *LNCS*, Tallin, Estonia, sep

2005.

[30] 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.

[31] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 329–338, Los Alamitos CA, March 2004. IEEE Computer Society Press.