Sub-method Structural and Behavioral Reflection

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Marcus Denker

von Deutschland

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 26.05.2008

Der Dekan:

Prof. Dr. P. Messerli

source: https://doi.org/10.24442/boristheses.1033 | downloaded: 26.4.2024

This dissertation is available as a free download from http://scg.unibe.ch

Copyright © 2008 Marcus Denker.

The contents of this dissertation are protected under Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page: creativecommons.org/licenses/by-sa/3.0/
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license): creativecommons.org/licenses/by-sa/3.0/legalcode

Published by Marcus Denker, Switzerland.
First Edition, 2008-05-08

Acknowledgments

First I want to thank my advisor Oscar Nierstrasz for the opportunity to join the Software Composition Group, and Stéphane Ducasse, who knew before me that I would come to Bern. Thanks to both of you for your advice and support!

I would also like to thank the other members of my Ph.D. committee: Robert Hirschfeld for many discussions and nice visits to Potsdam. Thanks to Pierre Cointe for reviewing the thesis, and Horst Bunke for accepting to chair the examination.

Special thanks goes to Éric Tanter, for great work to build on, for the great collaboration, many discussions and a very great stay in Chile.

Special thanks to my students: without you, it would have been impossible! To Philippe Marschall who fearlessly realized Persephone, Christoph Hofer for using Bytesurgeon for real, David Röthlisberger for his work on Geppetto and Anselm Strauss for going the aspect way.

Thanks to Frédéric Pluquet, Stefan Reichert, Adrian Lienhard, Lukas Renggli and Nik Haldiman for using Reflectivity. Special thanks to David for his work on Geppetto, first as a student and later colleague. Thanks to Mathieu Suen for collaborating on the context extension.

I want to especially thank the current and former members of the SCG: Orla Greevy, Adrian Kuhn, Adrian Lienhard, Lukas Renggli, David Röthlisberger, Toon Verwaest, Markus Gälli, Laura Ponisio, Tudor Gîrba, Nathanael Schärli, Alexandre Bergel, Michele Lanza, Gabriela Arévalo, Matthias Rieger, Iris Keller and Therese Schmid. It was and is a wonderful group!

Thanks to Martin von Löwis, Klaus Witzel, Michael Haupt and Hans Beck. Special thanks to my brother Christian for the cover design and Orla for reviewing the thesis and lots of support. Thanks to my old friend Jürgen for exploring Switzerland together, to Markus for a lot of Squeak fun.

Many, many thanks to my family! Thanks for your love and support:

Erika, Gerhard, Melanie, Volker, Hairi, Sonja, Tobias, Lena, Lotta, Christian and Susanne.

Abstract

Computational reflection is a fundamental mechanism in object oriented languages. Reflection has proved useful in many contexts, such as in the design of development environments, language extension, and the dynamic, unanticipated adaptation of running systems.

We identify three problems with the current approach to reflection in object oriented languages: partial behavioral reflection needs to be anticipated, structural reflection is limited to the granularity of a method, and behavioral reflection cannot be applied to the whole system.

To address these problems, we extend structural reflection to cover sub-method elements and present how sub-method structural reflection supports unanticipated partial behavioral reflection. We add the concept of context to represent meta-level execution and show how this allows behavioral reflection to be applied even to system classes.

We describe an implementation in Smalltalk. Benchmarks validate the practicability of our approach. In addition, we present an experimental evaluation in which we show how the system is used for dynamic analysis. We realize dynamic feature analysis by annotating the sub-method structure of the system directly to denote features instead of recording full execution traces.

Contents

Chapter 1

Introduction

1.1 Context

Software systems are increasing in size and complexity with the growth of the available computing resources like memory and processing power. Furthermore, applications are seldom deployed as standalone entities: typically they are embedded in a complex environment of interacting, interdependent systems. Consequently, the demands placed on software systems are changing.

A typical characteristic of an application running in a production environment is that is it difficult to shut down one application without causing adverse effects on the other applications. There are many complex and mission critical applications deployed in various domains, for example network management, air traffic control or banking. Halting a banking application for maintenance would have massive financial repercussions. In such cases, it is undesirable to shut down systems for maintenance. We need mechanisms that offer support for monitoring, analyzing or even modifying such systems while at the same time keeping them up and running.

Not only do the demands we put on the systems change. The demands we put on tools, environments and languages that are used to implement the systems are changing, too. We need to be able to build tools and develop new language features to satisfy the needs of the future. Some of the required functionalities have already been identified and some can be even realized within existing systems. However most of the functionality that will be required in the future is not yet known and thus cannot be anticipated. To address this, we need thus flexible systems that allow the

programer to experiment with new tools, environments and even new language features.

Reflection supports these needs. Reflection in programming languages is a paradigm that supports computations about computations, so-called *meta-computations*. Meta-computations and base computations are arranged in two different levels: the *meta-level* and the *base level* [119, 56]. Because these levels are causally connected, any modification to the meta-level representation affects any further computations on the base level [96]. In object-oriented reflective systems, the meta-level is formed in terms of meta-objects: a meta-object acts on *reifications* of program elements (execution or structure). If reifications of the *structure* of the program are accessed, then we talk about *structural reflection*; if on the other hand reifications deal with the *execution* of the program, then we refer to *behavioral reflection*.

Current object-oriented languages provide access to program structure reifications. In some cases, for example in static languages like Java and C#, these descriptions can be queried. Dynamic languages in addition offer support for changing the representation: they are *reflective*. Examples are CLOS [88], Ruby, Python and Smalltalk [48, 57]. In such languages, the representation of the program is causally connected to the system itself [97]. When the representation is changed, the running application changes, too and conversely. Structural reflection is an interesting basis for tools: Development environments can work directly on the representation provided by the language. An example for such a system is Smalltalk [57]. However, structural reflection does not provide a suitable representation for structure below the method level. For example in Smalltalk (the same holds in Java) the only entities supporting limited sub-method structural reflection are collections of bytes (from the compiled method) or characters (from the textual representation of method bodies).

There are many examples of systems proving some form of *behavioral reflection*. Realizing a fully behaviorally reflective system in a practical way, though, is not a trivial problem as it is particularly difficult to introduce behavioral reflective properties into a system without adversely affecting the performance. An interesting strategy to provide a practical realization of behavioral reflection is *partial* behavioral reflection [128]. Partial behavioral reflection restricts the introduction of reflective capabilities to the parts of a system where they are needed, thus not costing anything when these capabilities are not used. Partial behavioral reflection was realized for Java using bytecode manipulation.

1.2 Problems of Reflection

In this dissertation, we take a look at existing approaches to reflection and outline solutions for three shortcomings we identified.

Anticipation. *Partial behavioral reflection needs to be anticipated.*

To support the analysis of long running systems, we need to be able to install and remove for example profilers and tracing tools at runtime. In addition, we want to be able to install our tools only on those parts of the system that we are interested in. Thus we need fine-grained spatial and temporal control over behavioral reflection.

Partial behavioral reflection is in principle the answer to this problem. The idea of partial reflection is to be able to select where and when behavioral reflection is active. When not used, the reflective features should not have any impact on the performance of the system. It is important to note that this implies that the decision what to reflect on is done at runtime: at any point in time we want to be able to change where reflection is applied and where not.

The problem now is how to realize this in practice. We need to realize partial behavioral reflection in a way that the system does not need to be prepared up-front. But when we look at the existing realization, it does not allow the decision what to reflect on to be taken at runtime. The client is required to anticipate all future needs of reflection. The statically defined reflective features can be enabled at runtime. The slowdown is minimal when they are disabled. But the principle problem of *anticipation* exists.

This problem comes from the implementation substrate used (namely the static Java virtual machine). Nevertheless, realizing a completely dynamic version of partial behavioral reflection poses a problem to be solved.

We need *unanticipated partial behavioral reflection*.

Sub-method Structure. *Structural reflection is limited to the granularity of a method.*

Even in the case of languages that support structural reflection, the system does not provide a representation of the structure of methods themselves. As a consequence, structural reflection is only useful when we deal with program structure at the level of classes and whole methods. This is sufficient for example for a tool like a class browser that works at this level of abstraction. However, tools working on more fine-grained abstractions than a complete method (for example,

debuggers or tracing tools) are not provided by the system with a representation of sub-method structure.

In addition, partial behavioral reflection needs mechanisms to control program elements at a finer granularity than a method. Nevertheless, even systems that support fine-grained behavioral reflection, structural reflection does not provide mechanisms to reflect on sub-method elements of a program. Partial behavioral reflection is realized with ad-hoc bytecode manipulation. This leads to a number of problems:

Semantic mismatch. Concepts existing at the level of the language are not represented at the level of the bytecode.

Synthesized code. As the bytecode is transformed it is difficult to distinguish between synthesized code and base level code, and provide a mapping back to the program source.

Code quality. Bytecode level abstractions make it hard to generate optimized code

To address these problems, we need *sub-method reflection* to support behavioral reflection and tool-building.

Recursion Problem. *Behavioral reflection cannot be applied to the whole system.*

As soon as we introduce modified behavior using reflection at the level of system classes, we run into the *recursion problem:* meta-level executions have to make use of system-level classes to carry out their functionality, which in turn triggers meta-level functionality again [29]. In addition to the case of system classes, the recursion problem also occurs when we try to use a meta-object on itself. This happens for example when trying to analyze a reflection based analysis tool using itself. This limits the use of reflection in practice. We show that the problem is solved by modeling *meta-level execution* using context.

We state our thesis as follows:

Thesis:

To support unanticipated behavioral reflection, reflection needs to be extended with sub-method structure and with the concept of context.

This dissertation tackles the three problems step by step. We realize partial behavioral reflection in a dynamic programming language to achieve *unanticipated* partial behavioral reflection. This implementation serves as a case-study to show the problem of a missing sub-method representation for structure. We discuss sub-method structural reflection, provide a working

implementation and show how this is beneficial for building tools. The next step then is to go back to partial behavioral reflection and realize it on top of sub-method structural reflection. We discuss how this solves the problems identified with the bytecode-based solution: semantic mismatch, code quality and synthesized code. As a last step we solve the problem that behavioral reflection cannot be applied to the whole system. We present an example of the problem and show how to solve it by providing a representation for meta-level execution. The dissertation closes with a case study that shows the use of the complete system for dynamic analysis.

1.3 Contributions

The contributions of this dissertation are:

1. *Unanticipated Partial Behavioral Reflection.* We extend partial behavioral reflection as pioneered by Reflex [128] to support completely unanticipated use [40, 116]. This means that the program does not need to be prepared in any way, reifications can be introduced at runtime, and there is no slowdown in performance of the program when reflection is not active.

2. *Sub-Method Structural Reflection.* We extend the traditional model of structural reflection to cover sub-method elements [39]. The representation is based on an abstract syntax tree (AST) that can be annotated. We provide an extensible compiler framework for defining the semantics of annotations.

3. *Partial behavioral reflection based on annotations.* We realize unanticipated partial behavioral reflection using sub-method reflection. This solves the problems we found with bytecode-based approaches: semantic mismatch, synthesized code and code quality. In addition, the resulting system has good performance characteristics: both the selection of reified elements and the resulting code are improved compared to the classical bytecode based implementation.

4. *MetaContext and contextual meta-object activation.* We extend reflection with the concept of meta-context to represent meta-level execution [**?**]. We demonstrate how this solves the problem of *meta-object call recursion*. This allows behavioral reflection to be applied to the whole system, including system classes and the meta objects themselves.

5. *Feature Annotations* and the use of sub-method reflection for *dynamic analysis* [42, 43]. We describe the use of our framework for dynamic

analysis in general and present a more in depth case study for *dynamic feature analysis*.

6. *Reflectivity*, an implementation of both sub-method reflection and partial behavioral reflection in Squeak Smalltalk. The implementation requires no changes to the virtual machine and is nevertheless practically usable both considering memory usage and execution performance.

1.4 Structure of the Dissertation

Chapter 2 discusses the context of this work. After a brief overview of reflection in object oriented systems, we discuss the new demands of future systems in the context of reflection. We analyze three shortcomings of current reflective systems that are solved in this dissertation.

Chapter 3 focuses on unanticipated behavioral reflection. We give an overview of BYTESURGEON, a runtime bytecode transformation framework. Based on bytecode transformation, we realize the GEPPETTO framework for unanticipated partial behavioral reflection. Even though we succeed in providing unanticipated use, the decision to base the solution on bytecode transformation proves to be problematic. The role of this chapter is thus twofold: on the one hand, it provides a first step towards our goal (it solves the problem of unanticipated use). On the other hand, it serves as a case study to highlight the need for sub-method structural reflection.

Chapter 4 describes sub-method structural reflection. We analyze the problems of finding a suitable model for structural reflection that provides support for representing program structure below the method level. We describe how to support such a high-level representation on today's bytecode based languages. To validate sub-method reflection, we present TREENURSE, an inlinling framework that can replace typical bytecode manipulation frameworks, and TYPEPLUG, a pluggable type system.

Chapter 5 discusses how we provide partial behavioral reflection on top of sub-method reflection. We revisit the work presented in Chapter 3 and simplify the model. We present benchmarks to validate the improved realization of partial behavioral reflection.

Chapter 6 elaborates on the problem of meta-object call recursion and describe how we extend partial behavioral reflection with the notion of *context* to represent meta-level execution. We show how this solves the recursion problem.

Chapter 7 shows how to use annotation-based partial behavioral reflection

in the context of *dynamic analysis*. We discuss the use of annotations for feature-analysis and show a case study.

Chapter 8 presents conclusions. We describe the impact of our work and outline future work.

Appendix A describes the REFLECTIVITY system from a practical standpoint. We show how to install and use the system based on examples.

Appendix B provides a glossary of terms used in the thesis.

Chapter 2

Reflection: Context and Problems

2.1 Introduction

In this chapter, we discuss the context of our work. We first discuss practical examples and the requirements they imply for reflection. Then we give a brief overview of reflection in general. We discuss reflection in programming languages and the theoretical model of the reflective tower. After a short overview of reflection in object-oriented languages we elaborate on implementation aspects.

We discuss three shortcomings of reflection in detail: anticipation, submethod structure and the recursion problem. We finish with a roadmap of how this dissertation solves these problems.

2.2 Problems in Practice

2.2.1 Dynamic Analysis.

In recent years there has been a revival of interest in dynamic analysis [70]. 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 [64, 71, 135].

Dynamic analysis yields precise information about the runtime behavior of systems [3]. However, the task of writing tools to abstract runtime data is not trivial. Tool builders are faced with many options as there are numerous techniques that address the task of collecting runtime data.

Reflection is an interesting implementation technique for dynamic analysis and has been used in the past, for example to record traces [24, 74]. The benefit of using reflection for dynamic analysis is that we can for example introduce tracers at runtime and retract them without the need to restart the system. In addition, we do not need to run the program in a special environment. For example, there is no need for a special virtual machine.

Partial behavioral reflection is especially suited as a basis for dynamic analysis [42], as we can precisely select what to reflect on and what data of the application to pass to the meta-object. To support dynamic analysis for long-running systems, we need to make sure that when not used, the reflective feature should not cost anything. We need to be able to introduce the reflective features on-demand at runtime, we need *unanticipated* partial behavioral reflection.

An important property of any technique for dynamic analysis is what parts of the system it can be applied on. A problem with reflection based approaches is that we face the *recursion problem* as soon as we reflect on core system features, like Numbers or Arrays. These are used in the code of the tracer itself, which leads to the problem of meta-object call recursion. In practice, reflection-based dynamic analysis is thus restricted to application level code. In addition, when implementing tools leveraging behavioral reflection, it would be interesting to analyze these tools themselves. This *dynamic meta-level analysis* is not practical to realize with current reflective systems. We will show that the cause for both problems is a missing representation of *meta-level execution*.

Requirements: (i) *unanticipated partial behavioral reflection* and (ii) *we need to apply behavioral reflection to the whole system*

2.2.2 Integrated Development Environments.

Development environments (IDEs) and tools can benefit from reflection: any IDE needs a representation of the program to work with. This representation is then used to drive code browsers, code highlighting and refactoring functionality. But most IDEs provide a distinct representation separate from the reflective representation already made available by the language [130]. With a sufficiently powerful reflective model, no special IDE data structures for the language are needed. Instead, the tools can use

the one representation that is the reflective model of the language itself.

In the light of long-running systems this is especially interesting as the boundaries between development and deployment start to blur. An IDE working on the reflective structure of a system facilitates development at runtime. A prime example for such a system is Smalltalk [60]. In the Smalltalk IDE, the code browser is just a very thin user interface layer that directly modifies the reflective structure of the language. Classes and methods are objects, the classes are responsible to compile and install new code as methods.

But even in systems where the IDE uses structural reflection, reflective representation of the system does not provide any support for the structure of methods themselves besides the low-level bytecode and source code. A representation of complete program structure would be interesting for any IDE feature working on sub-method abstraction, for example for debuggers, profilers or code refactoring tools.

For tools working on the level of classes and whole methods (*e.g.*, a class browser), structural reflection eases tool-building as all tools use the same representation [130]. The system should provide the same for *sub-method* structure.

Requirement: *sub-method reflection*.

2.2.3 Language Experiments

Reflection provides means to experiment with new language features without the need to develop completely new languages. For example, behavioral reflection has been used to introduce new language features in several languages, for instance multiple inheritance[16], distribution [5, 102], instance-based programming [4], active objects [25], concurrent objects [134], futures [105] and atomic messages [57, 99], as well as backtracking facilities [91]

An interesting example is the existing structural reflection found in Squeak Smalltalk. In the past, Squeak has already provided a platform for many experiments with new language features [6]. Examples of such language experiments include:

Traits are building blocks that can be composed to classes [52, 11]. Traits have been realized in Squeak without the need to change the virtual machine or the grammar of the language by using reflection.

Classboxes [10, 9, 8, 7] provide a module system that supports local rebinding.

Changeboxes [41, 136] provide a mechanism for encapsulating change as a first-class entity in a running software system. Changeboxes provide for running different versions of code in the same system.

All these examples use either the standard reflective facilities of Smalltalk [113] or those added to the Squeak dialect, for example the ability to replace any method by an object that interprets the message sent [6]. These examples show that limited reflective properties are already very useful in supporting language experiments. Improving reflective features will help to make language experiments even easier.

Requirements: (i) *sub-method reflection*, (ii) *unanticipated behavioral reflection*.

2.2.4 Three Requirements for Reflective Systems

Reflection will play a crucial role to enable us to build both the languages and tools for the future. We have already seen many examples of how reflection has been useful for these kinds of experiments in the past. At the same time, we found three shortcoming that we plan to solve in this dissertation:

- Analyzing a long-running systems demands *unanticipated use* of partial behavioral reflection.
- We need to extend structural reflection with *sub-method* support to serve as a basis for tool building and behavioral reflection.
- We need to solve the *recursion problem* to be able to use tools based on behavioral reflection on the complete system including system classes and the code of the tool itself.

Before we discuss these requirements in detail, we give a brief overview of reflection.

2.3 Reflection: A Brief Introduction

We first discuss reflection in programming languages in general, then reflection in object-oriented languages in particular. This section is kept short. For more information, we refer the reader to the literature.

Appendix B provides a glossary for easy reference.

2.3.1 Reflection in Programming Languages

In this section we define common terms that are used when describing reflective systems, and we provide a definition for *reflective systems*. The definition follows the paper of Maes [97]:

> *A* **reflective system** *is a system which incorporates causally connected structures representing (aspects of) itself [97].*

This definition is very general. Programming languages constitute just one example of a reflective system. In reflective programming languages, the system incorporates a self-representation of the language. The requirement of *causal connection* is fundamental:

> *A system is said to be* **causally connected** *to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect of the other [97].*

A reflective language thus has a representation of its own structure and behavior available from within. The representation changes if the language changes and vice versa. It is always in sync with the system itself. Therefore, the representation can be queried and it can even be changed.

Many programming languages provide mechanisms to query a representation of the system. This is known as *introspection*. If a programming language provides mechanisms to change the representation of itself, this is referred to as *intercession*. Only when we can both query and change the representation, we call the system *reflective*.

The Reflective Tower

Reflection as a concept for procedural programming languages was first studied by Smith [119, 120, 45]. His theoretical model is that of meta-level interpretation. A language is interpreted by an interpreter, which is in turn interpreted by an interpreter. The result is an endless tower of reflective interpreters as shown in Figure 2.1.

A tower-of-interpreters model as such is not yet reflective, as the interpreter at one level above is not available for introspection or intercession to the program being interpreted. The ability to reflect on the interpreter is provided by special functions: a program may request code to be interpreted at the level of the interpreter and thus making a switch to a higher meta-level by calling a *reflective function*.

The mechanism for switching levels was further studied by Friedman

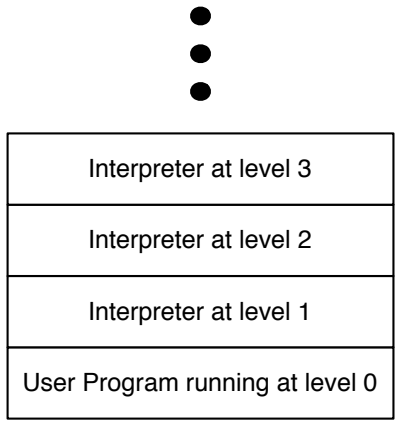

Figure 2.1: The Reflective Tower

and Wand [59]. They present a model with the goal of eliminating a tower-of-interpreters. They define what needs to happen for a level shift. They describe a two step process. The first is called *reification*:

> **Reification** *is the conversion of an interpreter component into an object which the program can manipulate. [59]*

The inverse function of reification is called *reflection*, which means installing the results of the meta-calculation back into the interpreter. Today the term *reification* is used in general to denote the creation of objects that represent concepts normally not accessible at a certain level. To *reify* something thus means to create an object for something that normally is not represented. An example for this is the concept of sending a message in an object-oriented language. For performance reasons, messages are not represented as objects.

2.3.2 Reflection and Objects: Meta-object Architectures

The main focus of Smith was the theoretical foundation of reflection, not actual practical reflective languages. As the reflective interpreter is too slow, other approaches are needed to provide reflective features in a language. A solution is provided by models for reflection that are based on language entities, not interpreter data structures. This view of reflection aligns very well with the object-oriented paradigm: in object-oriented reflective sys-

tems, the meta-level is formed in terms of meta-objects. A meta-object acts on *reifications* of program elements (execution or structure).

Computational reflection was first studied in object-oriented languages by Maes [97]. The 3-KRS system incorporated the idea of Smith into an object-oriented system. Here we reflect on object structure, rather than interpreter structure. Other examples are systems like Smalltalk where the reflective features are the result of the implementation strategy: Smalltalk is implemented to a large extent in Smalltalk, which results in reflective capabilities [57] as we can change the Smalltalk class structure at runtime.

Meta-object Protocols

The CLOS language developers pioneered the term *meta-object protocol* (MOP) to denote the reflective features of a language [87, 88]. As the meta-level in meta-object architecture is formulated in terms of objects, these so-called meta-objects provide a protocol for manipulation in the style of an Open Implementation [85, 86]:

> *"Metaobject protocols are* interfaces *to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language."* [88]

Structural and Behavioral Reflection

The literature distinguishes structural and behavioral reflection [56]: structural reflection deals with program structure while behavioral reflection deals with the execution.

Behavioral and structural reflection can be seen on the one hand as orthogonal concepts: a language can provide functionality for behavioral or structural reflection or both. On the other hand, they are connected: any change of structure leads to a change of behavior and any behavioral change needs to change structure at some level.

As a structural change can be used to change behavior, structural reflection can serve as the basis for behavioral reflection. One example for this is MethodWrappers [24], which allows methods to be wrapped to execute additional code before or after the method. Another example is Reflex [128] which realizes behavioral reflection by transforming bytecode. In Section 2.4 we discuss how to realize behavioral reflection using the structural reflection of Squeak Smalltalk.

The fact that a behavioral change leads to a structural change at some

level can lead to the problem that this structural change is visible to introspection. We will see that this is can lead to problems when realizing behavioral reflection using bytecode modification in Chapter 3.

Behavioral Reflection: Engineering the Meta-Level

Meta-objects are associated to base-level entities of the language. As there are different entities, there are many different possible ways of how to structure the meta-level.

Object. An example is the language 3-KRS [97]. Here meta-objects are associated per object, the behavior of an object is defined by its associated meta-object.

Class. All objects of one class share meta-object. As we have already a structural meta-object, the meta-class, this meta-object is reused as a behavioral meta-object. Examples are the CLOS MOP [88] and all other meta-class based models of behavioral reflection like MetaClassTalk [17].

Operation. McAffer suggests an *operational* decomposition of the meta-level [100]. Reifications represent *occurrences* of *operations* denoting the activity of the base program execution. Examples of operations are message sending, method execution, and variable access. An occurrence of an operation is a particular event, for example a particular sending of a message.

2.3.3 Partial Behavioral Reflection.

Partial behavioral reflection follows the operational decomposition of the meta-level. It allows for flexible engineering of the meta-level, making it possible to design a *concern-based* meta-level decomposition (*i.e.*, where one meta-object is in charge of one concern in the base application) rather than the typical *entity-based* meta-level decomposition (*e.g.*, one meta-object per object, or one meta-object per class). Hence it is possible to reuse or compose meta-objects of different concerns which greatly eases the engineering of the meta-level [100, 128]. In addition, *partial* behavioral reflection limits the performance impact by letting users precisely select what needs to be reified.

With partial behavioral reflection is possible to select exactly which operation *occurrences* are of interest, as well as *when* they are of interest. These spatial and temporal selection possibilities are of great advantage to limit costly reification. Furthermore, the exact communication protocol

between the base and meta-level is completely configurable: which method to call on the meta-object, pieces of information to reify, etc. We discuss partial behavioral reflection in detail in Section 2.4.

2.3.4 Implementing Behavioral Reflection

Interpreter-based Reflection

The tower-of-interpreters, or in general a meta-circular interpreter [1] is in principle an implementation strategy for reflection. The problem is practicability: such a system of multiple interpreter-stages is hard to realize without adversely affecting performance.

A special case of the reflective interpreter is an interpreter that provides an open implementation [85, 86]. An example is the work of Steyaert [123]. Here the interpreter provides a pre-defined subset of changeable features.

Language-based Reflection

In object-oriented languages, reflection is defined in terms of objects. Entities of the language, for example classes are controlled by meta-objects. In these systems, the language is structured in a way that allows behavior to be changed. Default behavior might be defined in the virtual machine for performance reasons. Examples here are 3-KRS [97], and meta-class based reflective systems like MetaClassTalk [17] where the meta-class of a class determines the behavior of for example variable access. The meta-class can be changed by inheritance or mixin-composition.

There is a tradeoff between generalization and performance: when we support many aspects of the language to be reflectively changed (*e.g.*, method lookup, instance variable storage), the performance suffers due to the needed indirections and reifications. As the virtual machine is fixed, the degree of reflection supported cannot be changed without changing the virtual machine or resorting to code generation approaches as described in the next section.

Code Transformation

In current mainstream languages, having a closed interpreter (or even no interpreter at all) is the normal case. The absence of an open interpreter does not prevent reflection: we can transform the program instead of the interpreter to realize behavioral reflection.

Historically, this approach was pioneered in the context of static compile

time MOPs [27], it was later applied to bytecode-based systems [30, 131] as this provides a very practical approach for realizing behavioral reflection for languages bytecode interpreting virtual machines.

We have already discussed that structural reflection can be used to provide behavioral reflection. The transformation approach in combination with structural reflection provides a powerful substrate for realizing behavioral reflection: transformed code can be installed in the system by using structural reflection at load-time or runtime. Practically, reflection is realized by introducing small code snippets, called *hooks*, in the places where meta-objects are supposed to be called. The positive aspect of code transformation is that it is very general, we can do most things that an interpreter-based solution can do. One can see it as a form of ad-hoc partial evaluation of the tower-of-interpreter. Even fine-grained and partial behavioral reflection can be realized, as we can control transformation down to the level of operations. One example for this is *partial behavioral reflection* as realized in Reflex [128].

There are multiple problems with the code-transformation approach to behavioral reflection. To support run-time transformation and thus *unanticipated use* of behavioral reflection, structural reflection needs to be powerful enough to replace the methods of a class at runtime.

When we change program structure, these changes are visible via the introspective API of the language. When transforming code on the level of bytecode, in addition there is the problem of having to deal with a low-level representation: concepts that have a representation in the language might already be optimized away (*e.g.*, jumps for control-structures instead of message sends).

Another problem is the missing interpreter scope. The transformation is global, which leads to the problem of endless meta-call recursion, preventing us from using behavioral reflection on the whole system.

2.4 Problem 1: Anticipation

Partial behavioral reflection needs to be anticipated.

We choose the Smalltalk [62] dialect Squeak [82] to implement a dynamic approach to reflection which supports *unanticipated partial behavioral reflection* (UPBR), because Squeak represents a powerful and extensible environment, well-suited to implement and explore the possibilities of UPBR. Before presenting our proposal, we discuss the current situation of reflective support in standard Smalltalk-80 as well as in the MetaclassTalk extension [17, 19, 21]. We also discuss closely related proposals formulated

in the Java context, both for unanticipated behavioral reflection and for partial behavioral reflection.

Smalltalk is one of the first object-oriented programming languages providing advanced reflective support [113]. The Smalltalk approach to reflection is based on the meta-class model and is thus inherently structural [56]. A meta-class is a class whose instances are classes, hence a meta-class is the meta-object of a class and describes its structure and behavior. In Smalltalk, message lookup and execution are not defined as part of the meta-class however. Instead they are hard-coded in the virtual machine. It is thus not possible to override in a sub-meta-class the method which defines message execution semantics. While not providing a direct model for behavioral reflection, we can nevertheless change the behavior using the message-passing control techniques [48], or method wrappers [24]. Also, the Smalltalk meta-model does not support the reification of variable accesses, so the expressiveness of behavioral reflection in current Smalltalk is limited.

Although reflection in Smalltalk can inherently be used in an unanticipated manner, the existing *ad hoc* support for behavioral reflection in Smalltalk is not efficient and does not support fine-grained selection of reification as advocated by *partial* behavioral reflection (PBR) [128]. For both reasons (limited expressiveness and lack of partiality), we have to extend the current reflective facilities of Smalltalk.

Extended Behavioral Reflection in Smalltalk: MetaclassTalk

MetaclassTalk [17, 19, 21] extends the Smalltalk model of meta-classes by actually having meta-classes effectively define the semantics of message lookup and instance variable access. Instead of being hard-coded in the virtual machine, occurrences of these operations are interpreted by the meta-class of the class of the currently-executing instance. A major drawback of this model is that reflection is only controlled at class boundaries, not at the level of methods or operation occurrences. This way MetaclassTalk confines the granularity of selection of behavioral elements towards purely structural elements. As Ferber says in [56]: “metaclasses are not meta in the computational sense, although they are meta in the structural sense".

Besides the lack of fine-grained selection, MetaclassTalk does not allow for any control of the protocol between the base and the meta-level: it is fixed and standardized. It is not possible to control precisely which pieces of information are reified: MetaclassTalk always reifies everything (*e.g.*, sender, receiver and arguments in case of a message send). Recent implementations of the MetaclassTalk model limit the number of effective reifications by only calling the meta-class methods if the metaclass indeed

provides changed behavior. But even then, once a metaclass defines custom semantics for an operation, all occurrences of that operation are reified in all instances of the class. Hence MetaclassTalk provides a less ad-hoc means of doing behavioral reflection than standard Smalltalk-80, but with a very limited support for partial behavioral reflection.

Unanticipated Behavioral Reflection: Iguana/J

Iguana/J is a reflective architecture for Java [108] which supports unanticipated behavioral reflection and a limited form of partial behavioral reflection.

With respect to unanticipated adaptation, with Iguana/J it is possible to adapt Java applications at runtime without being forced to shut them down and without having to prepare them before their startup for the use of reflection. However to bring unanticipated adaptation to Java, Iguana/J is implemented in form of a dynamic library integrated very closely with the Java virtual machine via the Just-In-Time (JIT) compiler interface [108]. This means that the Iguana architecture is not portable between different virtual machine implementations: *e.g.*, the JIT interface is not supported anymore on the modern HotSpot Java virtual machine. Conversely, we aim at providing UPBR for Smalltalk in a portable manner, in order to widen the applicability of our proposal.

With respect to partiality, Iguana/J supports fine-grained meta-object protocols (MOPs), offering the possibility to specify which operations should be reified. However, precise operation *occurrences* of interest cannot be discriminated, nor can the actual communication protocol between the base and meta-level be specified. This can have unfortunate impact on performance, since a completely reified occurrence is typically around 24 times slower than a non-reified one [108].

Partial Behavioral Reflection: Reflex

A full-fledged model of partial behavioral reflection was presented in [128]. This model is implemented in Reflex, for the Java environment.

Reflex fully supports partial behavioral reflection: it is possible to select exactly which operation *occurrences* are of interest, as well as *when* they are of interest. These spatial and temporal selection possibilities are of great advantage to limit costly reification. Furthermore, the exact communication protocol between the base and meta-level is completely configurable: which method to call on the meta-object, pieces of information to reify, etc. The model of *links* adopted by Reflex, which consists of an explicit binding of a cut (set of operation occurrences) and an action (meta-object), also gives

total control over the decomposition of the meta-level: a given meta-object can control a few occurrences of an operation in some objects as well as some occurrences of other operations in possibly different objects.

Hence meta-level engineering is highly flexible, the limitation of Reflex however lies in its implementation context: being a portable Java extension, Reflex works by transforming bytecode. Hence, although reflective behavior occurs at runtime, reflective needs have to be anticipated at load time. This means that Reflex does not allow a programmer to insert new reflective behavior affecting already-loaded classes into a running application. Instead, the programmer is forced to stop the application, define the reflective functionality required and to reload the application to insert this meta-behavior. Links can be deactivated at runtime, but at a certain residual cost, because the bottom line in Java is that class definitions cannot be changed once loaded.

2.5 Problem 2: Sub-method Structure

Structural reflection is limited to the granularity of a method.

The second problem identified is the lack of a representation of sub-method structure. Having the structure of a method described as part of the structural reflective capabilities of a language has many uses. We highlight two reasons for sub-method structure: (i) sub-method reflection provides a basis for tool-building, and (ii) the missing structural representation for methods complicates the realization of partial behavioral reflection.

Problems of Bytecode-based Behavioral Reflection

Partial behavioral reflection has been realized both for Java and Smalltalk by using bytecode transformation.

Behavioral reflection needs some way to deal with sub-method elements, as typical behavioral entities are for example variable access or message sends. Nevertheless, even systems that support behavioral reflection, structural reflection does not provide mechanisms to reflect on sub-method elements of a program. Partial behavioral reflection has been realized both for Java and Smalltalk by using bytecode transformation. The low-level nature of bytecode results in a number of problems:

Semantic mismatch. Bytecode is a representation optimized for execution. Thus there are some optimizations made when generating bytecode, for example in Smalltalk, control structures are not realized with message sends. Instead, bytecode has instructions to encode jumps.

Concepts existing at the level of the language are not represented at the level of the bytecode.

Code quality problem due to low level model. Transformations at the level of bytecode make it difficult to generate optimized code.

Problem with synthesized elements. To realize behavioral reflection via any form of code transformation, we modify the structure of the program. This is problematic, as these changes are visible through structural introspection. It is, for example, hard to distinguish between synthesized code and base level code.

A Structural Representation for Tools

Tools that work on sub-method abstractions are common. Examples are trace-tools, debuggers, profilers or refactoring tools. For tools working on the level of classes and whole methods (e.g. class browser), structural reflection eases tool-building as all tools use the same representation. The system should provide the same for *sub-method* structure. The representation should be

- causally connected and well integrated in the system.
- persistent and extensible to allow tools to communicate.
- reasonably compact with minimal performance impact.
- offer a high-level abstraction to the developers.

2.6 Problem 3: Recursion

Behavioral reflection cannot be applied to the whole system.

When we deploy behavioral reflection to analyze a system, for example with a reflective profiling tool, we soon run into problems. Any reflective tool will use the basic system library at runtime. If we now introduce calls to the profiler in the library code using reflection, the result will be an endless loop that finally leads to a system crash. This means that we cannot apply behavioral reflection to any system library or to any other code that is executed as part of the meta-object.

To enable reflection in mainstream languages, the tower of interpreters is replaced with a reflective architecture [97] where meta-objects control the different aspects of reflection offered by the language. Meta-objects provide

the implementation of new behavior which is called at certain defined places from the base system. It is important to note that meta-objects are not special objects. The execution of code as part of a meta-object is not different to any execution occurring in the base level application. As both base and meta-computations are handled the same way, we are free to call any part of the base-system in the meta-level code.

This means that meta-level code can actually trigger again the execution of meta-level functionality. There is nothing to prevent the meta-level code to request the same code to be executed again, leading to an endless loop resulting in a system crash. This is especially a problem when reflecting on core language features (*e.g.*, the implementation of Arrays or Numbers) since chances are high that such features are used to implement reflective features themselves. These cases of spurious endless recursion of meta-object calls have been noted in the past in the context of CLOS [29].

The ability to reflect on system classes is especially important when using reflection for dynamic analysis. A tracing tool that is realized with reflection should be able to trace a complete run of the system, not only the application code. In addition to the problem of recursion, a tracer has the problem of recording the execution of trace code itself.

If we go back to the infinite tower (the origin of meta-level architectures) we can see that here these problems do not exist by construction: *going meta* means jumping up to another interpreter. A reflective function is always specific to one interpreter. As a function that is reflective at a meta-level I_n is not necessarily reflective in I_{n+1}, the problem of infinite recursion does not happen.

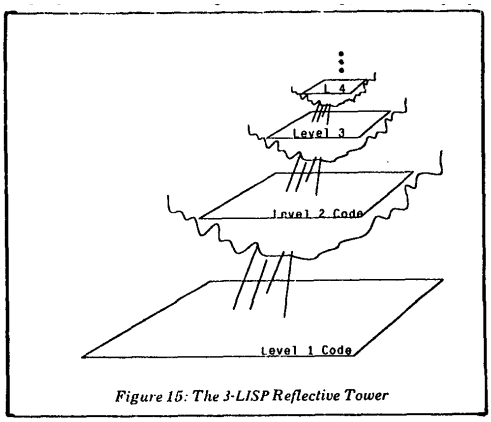

Figure 2.2: The 3-Lisp reflective tower from [119]

An important question then is the difference between the meta-object and interpreter/infinite tower approach. The *metaness* in the case of the tower is defined by the interpreter used. The interpreter forms a context that determines if we are executing at the base level or at the meta-level. Calling reflective functionality (so called reification) is always specific to one interpreter. The meta-object approach now in contrast is lacking any mechanism to specify this contextual information: when executing a meta-level program, in a meta-object based reflective system, we lack the information that this program is executing at the meta-level. In addition, all reifications are globally active: we cannot only trigger meta-object activation when executing base level code. The research question is thus how to incorporate the infinite tower property of explicitly representing the execution context into meta-object based architectures.

2.7 Summary and Roadmap

We have presented a short overview of reflection in general and an overview of reflection in object-oriented system in particular. We then discussed three shortcomings identified in existing reflective systems:

1. **Anticipation.** Partial behavioral reflection needs to be anticipated.
2. **Sub-method Structure.** Structural reflection is limited to the granularity of a method.
3. **Recursion Problem.** Behavioral reflection cannot be applied to the whole system.

Roadmap

We now solve these three problems step-by-step

Chapter 3. We realize partial behavioral reflection in Smalltalk. As is it based on runtime bytecode transformation, it solves the problem of *unanticipated use*. In addition, this chapter shows the need for a better representation of sub-method structure.

Chapter 4. We discuss *sub-method reflection*, show how to realize it in bytecode-based languages and present examples of how sub-method reflection supports tool building.

Chapter 5. With sub-method reflection, we can take a second look at partial behavioral reflection. Based on sub-method reflection, the new system does not exhibit the problems found in the bytecode based solution.

Chapter 6. We cannot apply our reflection framework to the whole system. We analyze the problem and solve it by extending partial behavioral reflection with the notion of *meta-level execution*.

Chapter 7. We provide a case-study of how to use the resulting system for dynamic analysis.

Chapter 3

Unanticipated Partial Behavioral Reflection

3.1 Introduction

In this chapter we present our first solution to solve the problem of anticipation for partial behavioral reflection. Both partial behavioral reflection (PBR) and unanticipated behavioral reflection (UBR) have been realized in the past, we combine both to provide *unanticipated* partial behavioral reflection (UPBR).

We first give an overview of BYTESURGEON, a bytecode transformation framework. Based on bytecode transformation we show how to realize *unanticipated partial behavioral reflection*.

The research shown in this chapter focuses just on achieving unanticipated use of partial behavioral reflection, but at the same time this first solution reveals that we need a better representation for sub-method structure. Thus the role of this chapter is twofold: first, it presents a step towards the solution by solving the problem of unanticipated partial behavioral reflection. But second, it makes the problems resulting from a missing sub-method structure more apparent:

- Semantic mismatch.
- Problem with synthesized code.
- Code quality problem due to low level model.

We discuss these problems in detail in Section 3.4.

3.2 ByteSurgeon: Bytecode Transformation

3.2.1 The Need for Bytecode Manipulation

Code can be transformed either directly as text (concrete syntax) or as an abstract syntax tree (abstract syntax). Furthermore, in language environments where source code is compiled to an intermediate bytecode language which is abstract enough, bytecode transformation is an interesting approach.

Disadvantages of AST Transformation

Transforming source code at the concrete syntax level is typically avoided because of the lack of structure and abstraction at the text level. Transformation of abstract syntax trees (ASTs) is much more appropriate, but still suffers from a number of limitations:

No original language warranty. Many mainstream languages today, such as Java, Squeak and C#, are based on a virtual machine executing bytecodes, and these virtual machines are actually used as the execution engines of various languages, other than the "original" ones. For instance, for the Croquet environment [121], a number of experimental scripting languages have been developed, among them languages similar to JavaScript and LOGO. Another example is the Python language, which can be compiled to Java bytecode [84]. To provide practical performance, these languages come with their own custom compiler that produces bytecode for a production-quality virtual machine. Therefore a code transformation tool working at the AST level rebuilding the AST from bytecode would require a custom decompiler. On the other hand, working on bytecode, although lower-level than AST, makes it possible to uniformly apply transformations even in the presence of non-original languages.

Recompiling is slow. Finally, transforming source code means that a compiling phase is necessary afterwards to regenerate bytecodes. Recompilation is a slow process, much slower than manipulating bytecode; benchmarks of Section 3.2.5 validate this statement.

3.2.2 Bytecode Transformation Approaches

A wide variety of tools have been proposed that rely on bytecode transformation. Surprisingly, most of these tools have been implemented for Java, and we are aware of very few related proposals in the Smalltalk world.

Java and Bytecode Transformation

The Java standard environment only allows for bytecode transformation at load time. At runtime, it is only possible to dynamically generate new classes from scratch, not to modify existing ones. These restrictions have been somehow relaxed in the context of the JVM debugger interface (JDI) [83], but relying on the debugger interface is not reasonable in a production environment. Furthermore, the capabilities of class reloading are strongly limited as, for instance, new members cannot be added to classes. Using load-time transformation in Java also raises a number of subtle issues related to class loaders and the way they define namespaces in Java [92].

Level of Abstraction

The experience gained with Java bytecode transformation tools brings a number of insights that ought to be considered when designing a new framework. The most fundamental one is that of the level of abstraction provided to programmers.

Tools like BCEL [35] and ASM [26] strictly reify bytecode instructions: as a consequence, users have to know the Java bytecode language very well and have to deal with low-level details such as jumps and alternative bytecode instructions (a Java method invocation can be implemented by several bytecode instructions, depending on whether the invoked method is from an interface, is private, etc.).

In contrast, Javassist [28] and Jinline [129] focus on providing *source code level abstractions*: although the actual transformation is performed on bytecode, the API exposes concepts of the source language. This is highly profitable to end users. In its latest version [30], Javassist even offers a lightweight online compiler so that injected code can be specified as a source code string. The Javassist compiler supports a number of dedicated meta-variables, which can be used to refer to the context in which a piece of code is injected.

As a matter of fact, bytecode-level manipulation is more complex than source-level manipulation because of the many low-level details one needs to deal with. However, working at the bytecode level also makes it possible to express code that is not directly expressable in the source language(s). This dilemma basically motivates the need for both APIs, as is done in Javassist: a high-level API provides source-level abstractions, and a low-level API provides bytecode-level abstractions.

3.2.3 BYTESURGEON: Overview

Proposals for Smalltalk

To the best of our knowledge there is no general-purpose bytecode manipulation tool for a Smalltalk dialect. AOStA [104] is a bytecode-to-bytecode translator that aims at providing higher-level, transparent, type-feedback-driven optimizations. It was not thought to be open to end users for bytecode manipulation[1]. Method wrappers [24] make it possible to wrap a method with before/after code. They are very fast to install and remove, as they do not need to parse bytecode or generate methods, but are not a general-purpose transformation tool. Several extensions actually need more power than just before/after control. AspectS [77] has been recently proposed as an aspect-oriented interface to the reflective capabilities of Smalltalk combined with method wrappers (to implement before/after advices). AspectS is actually a tool that would much profit from BYTESURGEON, as it would significantly raise its expressive power.

BYTESURGEON is our library for runtime program transformation in Smalltalk, currently implemented in the Squeak environment. BYTESURGEON complements the reflective abilities of Smalltalk [113] with the possibility to instrument methods, down to method bodies.

Smalltalk provides a great deal of structural reflection: the structure of the system is described in itself. Structural reflection can be used to obtain the object representing any language entity. For instance, the global variable Example stands for the class (the object representing the class) Example, and the object describing the compiled method aMethod in class Example is returned by the expression Example>>#aMethod.

Dynamically adding instance variables and methods to an existing class is fully supported by any standard Smalltalk environment. However the structural description of a Smalltalk system stops at the level of methods: a compiled method cannot be reflected upon. BYTESURGEON adds support for both introspection and intercession of compiled methods at the bytecode level.

Introspecting Method Bodies

Let us first see how BYTESURGEON is used to introspect method bodies. The following code statically counts the number of instructions that occur in all methods of the class Example:

[1] Actually, BYTESURGEON could profitably use AOStA for its backend, but this study is left as future work.

```
InstrCounter reset.
Example instrument: [ :instr | InstrCounter increase ]
```

The instrument: method is implemented in class Behavior. As a parameter it is given a block of standard Smalltalk code that takes one argument. This block is an *instrumentation block*: for each instruction within all methods of the class, the instrumentation block is evaluated with a reification of the current instruction as parameter. We will see later what an instruction reification is. For now, it suffices to say that for each instruction, a global counter is increased.

There are variants of the instrument: method for each particular language operation: constant, variable access, read and store and message sending. For instance, instrumentSend: only evaluates the instrumentation block upon occurrences of the message send operation. Besides sending the instrumentation message to a class, thereby affecting all its methods, we can send it to a single method:

```
SendMCounter reset.
(Example>>#aMethod) instrumentSend: [ :send | SendMCounter increase ]
```

Reification of Language Operations

Instructions in a method body are static occurrences of the operations of a language. BYTESURGEON supports message send, access to instance variables and local variables, and constants. The structural model representing language operations is shown on Figure 3.1[2]. This structural model is bytecode-based. It does not encode as much information as an AST does, *e.g.*, it is not possible to extract, from an IRSend, the instructions that correspond to the arguments of the send. This is a limitation of bytecode-based transformation as opposed to AST-based transformation.

When sending an instrumentation message (*i.e.*, instrument:, instrumentSend:) reification of instructions are built, as instances of the appropriate class in the hierarchy, and passed to the instrumentation block. The instrumentation block can then introspect and change them. For instance, the following piece of code prints the selector of each message send occurring within Example>>#aMethod:

```
(Example>>#aMethod) instrumentSend: [ :send |
                                Transcript show: send selector printString; cr]
```

[2]The isXXX methods (*e.g.*, isSend) are provided as a convenience to avoid the use of visitors and double dispatch.

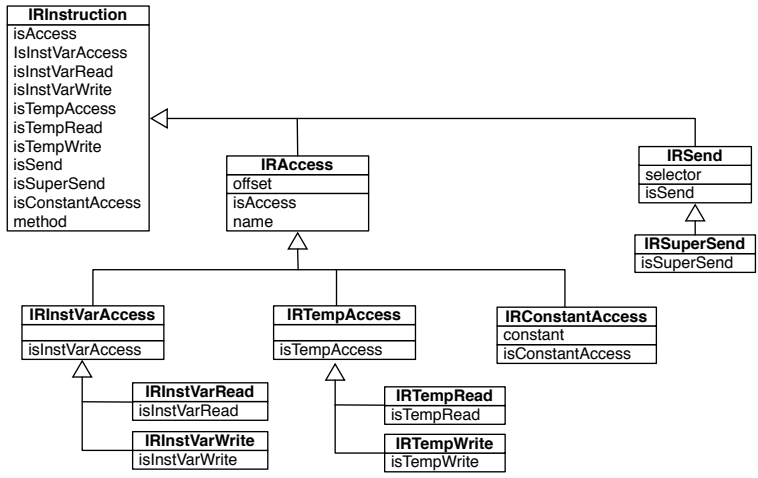

Figure 3.1: Structural model of instructions in BYTESURGEON.

Method Evaluation. A peculiar language operation is *message receive* (the callee-side equivalent of a message send). Actually, a message receive is realized by two operations: method lookup and method evaluation. Since we are working at the bytecode level, we do not have access to method lookup, only *method evaluation*. Rather than corresponding to a bytecode instruction inside a method body, method evaluation corresponds to a method body as a whole. Since BYTESURGEON treats all language operations in a uniform manner, methods have the same introspection and intercession interface as instructions.

Modifying Method Bodies

BYTESURGEON supports two ways of modifying method bodies: a bytecode-level manner, where the user directly specifies the required transformation in terms of bytecode representations, and a source-level manner, where the transformation is specified with a string of source code. We hereby only present the source-level API. The bytecode-level API is briefly mentioned in Section 3.2.4.

Similar to Javassist [30], BYTESURGEON provides an online compiler that makes it possible to specify code to be inserted as a string. The methods to insert code before, after and instead of an occurrence of a language operation are named respectively `insertBefore:`, `insertAfter:` and `replace:`. They take as argument the source code as a string, which is subsequently compiled by

the BYTESURGEON compiler, and the resulting code is inserted at the appropriate position. For instance, the following code inserts a call to the system beeper before each message send occurring within Example#>>aMethod:

```
(Example>>#aMethod) instrumentSend: [ :send | send insertBefore:  'Beeper beep' ]
```

The code string can contain any valid Smalltalk code[3], plus two kinds of special variables: *user-defined variables* to refer to statically-available information, and *meta-variables* for runtime information.

Accessing Static Information: User-defined Variables

Statically-known information about an instruction can be used in the construction of the string. For instance, the following example records the name of selector of each message send occurring at runtime:

```
(Example>>#aMethod) instrumentSend: [ :send |
    send insertAfter: 'Logger logSend:' , send selector printString]
```

Here we query the objects describing the message send operations for the name of the message sent. To ease the construction of the string and avoid hard-to-understand string concatenation, BYTESURGEON makes it possible to define custom variables with the syntax $<$: #$variable$$>$, and giving a list of associations from variable names to object references[4]:

```
(Example>>#aMethod) instrumentSend: [ :send |
    send insertAfter: 'Logger logSend: <: #sel >' using: { #sel –> send selector } ].
```

Accessing Runtime Information: Meta-variables

The online compiler of BYTESURGEON also supports a number of predefined meta-variables that refer to information available at runtime, such as the receiver of a message send (Figure 3.2). Meta-variables are an essential part of the expressiveness of a good bytecode transformation framework. The exact set of available meta-variables depends on both the operation selected —in the case of a message send, meta-variables are provided to refer to the sender, the receiver and the arguments— and the transformation to perform —when inserting after, it is possible to access the result—. Meta-variables are denoted by the $<$meta: #$variable$$>$ construct. For instance,

[3] self, super and thisContext have their usual meaning, knowing that this code will be evaluated in the place where it is inserted.

[4] This is a limited sort of quasi-quoting *a la* Scheme; supporting true quasi-quoting (with no needs to specify manually the associations) is left as future work.

Operation	Meta-variable	Description
Message Send/ Method Evaluation	<meta: #arguments>	arguments as an array
	<meta: #argX>	X^{th} argument
	<meta: #sender>	sender object
	<meta: #receiver>	receiver object
	<meta: #result>	returned result (after only)
Temp/InstVar Access	<meta: #value>	value of variable
	<meta: #newvalue>	new value (write only)

Figure 3.2: Meta-variables supported by BYTESURGEON.

the following code replaces each message send with a call to a dispatcher meta-object in charge of the actual method lookup [56, 48]:

```
(Example>>#aMethod) instrumentSend: [ :send |
   send replace: 'CustomDispatcher send: <: #selSymbol>  to: <:meta: #receiver> with:
      <:meta: #arguments>'
      using: { #selSymbol –> send selector printString } ].
```

The BYTESURGEON compiler takes care of generating the code to access the runtime information denoted by the meta-variables, by adding a preamble before the inlined code. The runtime overhead due to preambles motivated us to maintain a special syntax for meta-variables (meta), to raise the attention of users that these variables should be used conscientiously.

Altering Method Evaluation. To support transformation of method evaluation, method objects also support the insertBefore:, insertAfter: and replace: messages. As an example, the following code inserts a trace before each evaluation of a method in class Example:

```
Example instrumentMethods: [ :m |
   m insertBefore: 'Logger logExec: <: #sel > ' using: { #sel –> m selector} ]
```

The meta-variables for method evaluation are the same as for message sending (see Figure 3.2). The following example uses a meta-variable to access the method evaluation result:

```
Example instrumentMethods:
  [ :m |  m insertAfter: 'Logger logExec: <: #sel > result: <meta: #result>'
            using: { #sel  –>  m selector]
```

3.2.4 Inside BYTESURGEON

We now give an overview of the implementation of BYTESURGEON, in particular the relationship to the closure compiler and the transformation process. The low-level transformation API is also discussed.

Squeak

BYTESURGEON is currently implemented in Squeak [82], an open source implementation of Smalltalk-80 [62]. Squeak is based on a virtual machine that interprets bytecodes. During a normal compilation phase, method source code is scanned and parsed, an abstract syntax tree (AST) is created and bytecodes are generated for the corresponding methods (Figure 3.3).

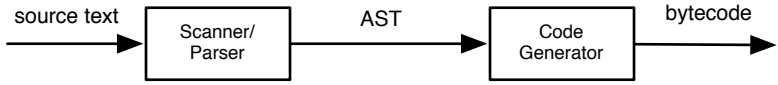

Figure 3.3: The standard Smalltalk compiler.

To implement BYTESURGEON in Squeak we could have directly worked at the level of bytecode. However, rewriting bytecode is tedious and error-prone for several reasons: the bytecode vocabulary is low-level, jumps have to be calculated by hand, and the expression of the context where bytecodes should be inserted is limited. Even simple modifications are surprisingly tedious to manage. Fortunately, a new compiler for Squeak, the *closure compiler* [72], has been recently proposed which offers a better intermediate bytecode representation.

The Closure Compiler and its Intermediate Representation

The closure compiler relies on a more sophisticated bytecode generation step (Figure 3.4): first an *Intermediate Representation (IR)* is created; then the IR is used to generate the real bytecode (the raw numbers).

The IR is a high-level representation of bytecode, abstracting away specific details: jumps are encoded in a graph structure, sequences of bytecode-nodes form a basic block, and jump-bytecodes concatenate these blocks to encode control flow. The main goal of IR is to abstract from specific bytecode encodings: for instance, although the bytecode for a program in Squeak is encoded differently than in VisualWorks, their IR is identical.

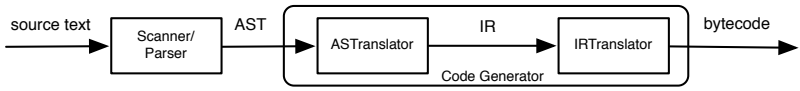

Figure 3.4: The closure compiler.

Using IR therefore makes the porting to other bytecode sets simple.

The closure compiler has a counterpart, the decompiler, which converts bytecode back to text. Here, the whole process works backwards: from bytecode to IR, from IR to AST, and finally from AST to text.

As motivated in Section 3.2.2, BYTESURGEON ought to offer adequate abstractions for both bytecode-level and source-level transformations. The IR of the closure compiler actually represents an excellent alternative for working at the bytecode level: it makes it possible to express code that is not directly obtainable from Smalltalk source code, while abstracting away many details.

All classes reifying instructions (recall Figure 3.1) are from the closure compiler IR. The low-level transformation API of BYTESURGEON is based on these classes. In addition to the classes reifying instructions which correspond to language operations, the IR includes classes reifying bytecode-only instructions: IRPop, IRDup, IRJump, IRReturn, etc.

Low-level Transformation API

We have used the high-level API of BYTESURGEON to specify transformations giving a string of source code, which may contain meta-variables to access dynamic information. The description of the new code to be inlined can also be done by directly editing the instruction objects for the IR hierarchy. In the following example, the selector of all sends of the message oldMessage:with: are replaced by sends of the message newMessage:with:, by using the selector: accessor of an IRSend object:

```
(Example>>#aMeth) instrumentSend: [ :send | send selector = #oldMessage:with:
        ifTrue: [ send selector: #newMessage:with: ] ].
```

The IRInstruction class can also be used as a factory to produce new objects describing bytecode. These objects can be used in replacement of the original instruction or be inlined before or after it. An alternative implementation of the code above is:

```
(Example>>#aMeth) instrumentSend: [ :send | send selector = #oldMessage:with:
        ifTrue: [ send replace: (IRInstruction send: #newMessage:with:)] ].
```

This implementation replaces the message send bytecode by a new one having a different selector. IRInstruction send: #newMessage:with: returns an object that describes a message send bytecode.

Specifying the transformation at the bytecode-level makes it possible to directly manipulate bytecode instructions and to easily specify transformations that are more complex to express with the source-level API. For instance, using the source-level API to change the selector of a message send, as done above, is done as follows:

```
(Example>>#aMeth) instrumentSend: [ :send | send selector = #oldMessage:with:
        ifTrue: [ send replace: '<meta: #receiver> perform: #newMessage:with:
                with: <meta: #arguments> ' ] ].
```

Apart from being slightly more verbose and relying on the use of the reflective message sending perform:with:, this approach requires the use of meta-variables, which are more costly due to the associated preambles that needs to be generated. Conversely, the low-level API makes it possible to do this transformation directly, without requiring runtime reification.

Implementation of Meta-variables

When BYTESURGEON instruments a method, the bytecode-to-IR part of the closure compiler generates the IR objects that are passed to the instrumentation block specified by the user. If the source-level API is used, then the code to be inserted is preprocessed to generate the IR nodes and to handle the meta-variables, if any. For meta-variables, a preamble code is generated to ensure that the expected values will be on the stack. Then, the preamble and code are inserted into the IR of the method. Finally, the IR-to-bytecode part of the closure compiler generates raw bytecodes and replaces the old method with the new, transformed version.

In the following we explain the implementation of meta-variables which reify runtime information. Let us consider the reification of the receiver of a message send.

Preambles. Squeak uses a stack-based bytecode, so all arguments for a message send are pushed on the stack before the send bytecode is executed: first the receiver, and then the arguments. For instance, the bytecode for the expression 3 + 4 is as follows:

```
pushConstant: 3
pushConstant: 4
```

```
send: +
returnTop
```

Consider that we now want to provide access to the receiver (3) via a meta-variable:

```
(Example>>#method) instrumentSend: [:send |
        send insertBefore: 'Transcript show:  <meta: #receiver> asString'].
```

We want to inline code that accesses meta-variables, for example we want to log the receiver of a message send. In the above case, it is clear that the object 3 is the receiver of the send. So one might think that we can statically transform the code. But this is not possible in general: the receiver itself could be the result of another expression, as for example in 1+2+3.

```
pushConstant: 1
pushConstant: 2
send: +
pushConstant: 4
send: +
returnTop
```

To be able to statically transform the code, we would need to decompile it from the bytecode representation back into an AST like form that models the expression as a tree. As this defeats the purpose of bytecode transformation, what is done is that we accept the fact that we only know the layout of the stack at runtime at any bytecode instruction. To make meta-variables available, we add code before the to-be-inlined code, the *preamble*, that stores this data in temporary variables. The temporary variables then can be easily accesses from within the inlined code.

To sum up, when we need to access any special data, like the receiver, we take advantage of our knowledge of the stack layout at that point: We need to add bytecode to store the necessary values, by popping them from the stack and storing them in additional temporary variables. In our example, we need the receiver. Since the receiver is deep in the stack, below the arguments, we also need to store the arguments in temporary variables, to be able to access them afterwards. In the case of before/after, it is also necessary to rebuild the stack.

The resulting bytecode for our example is as follows:

```
pushConstant: 3
pushConstant: 4
popIntoTemp: 0        "put argument in temp 0"
popIntoTemp: 1        "put receiver in temp 1"
pushLit: ##Transcript  "start of inserted code"
```

```
pushTemp: 1          "push receiver for printing"
send: asString
send: show:
pop                "end of inserted code"
pushTemp: 1          "rebuild the stack"
pushTemp: 0
send: +             "original code"
returnTop
```

To access all arguments as an array, the compiler generates code to create the array instance, to add arguments to it, and to store the array in a temporary variable.

For performance and space reasons, preamble generation needs to be optimized. First, the compiler only generates code for the meta-variables that are effectively used in the inlined code. For instance, if access to the arguments is not needed, then the array creation is avoided. The second important optimization is to reuse temporary variables. Indeed, there are potentially many operations for which we need to generate a preamble, in a single method. If we used new temporary variables for each, we would soon run out of temporary variable space (Squeak imposes a limit of 256 temporary variables per method). Therefore, BYTESURGEON remembers the original number of temporary variables and reuses the variables added for each preamble. This information is saved inside the compiled method object, so that reuse of variables works even if instrument: is executed several times on the same method.

Inlining code. Once the preamble is added, the code to inline can be inserted. First, the BYTESURGEON compiler generates the IR for the new code. For meta-variables, the compiler generates code that loads the corresponding temporary variables. The generated IR instructions are then added to the original IR of the method. If necessary, jump targets are adjusted and basic blocks renumbered. The new method IR is then given to the closure compiler, which generates the final raw bytecodes and installs the new method.

3.2.5 Validation

We now validate the interest of BYTESURGEON by showing how easy it is to implement two language extensions: method wrappers [24] and a simple runtime MOP for controlling accesses to instance variables. We complete this validation by reporting on performance measurements.

Method Wrappers

Method wrappers [24] wrap a method with before/after behavior. Wrapping a method is implemented by swapping out the compiled method by another one, valueWithReceiver:arguments: that calls the before method, then the original method, and finally the after method:

```
MethodWrapper>>valueWithReceiver: anObject arguments: args
    self beforeMethod.
    ^ [clientMethod valueWithReceiver: anObject arguments: args]
        ensure: [self afterMethod]
```

The BSMethodWrapper class contains the logic to install an instance of itself as a method wrapper, with empty before/after methods.

To define a wrapper, a subclass should be created, specifying the before/after methods. For instance, class CountingMethodWrapper wraps a method to count invocation of calls to a given method:

```
BSMethodWrapper subclass: #CountingMethodWrapper
  instanceVariableNames: 'count'...

CountingMethodWrapper >> beforeMethod
  self count: self count + 1
```

To count the invocations on a method, we install the wrapper:

```
wrapper  := CountingMethodWrapper on: #aMethod inClass: Example.
wrapper install.
```

The installation of a method wrapper consists in first decompiling the before/after methods to IR (ir), stripping the return at the end (strip), then replacing all self references to refer to the wrapper (replaceSelf:), and finally inlining the before/after methods (insertBefore:after:):

```
BSMethodWrapper>>inlineBeforeAfter
    | before after |
    before := (self class lookupSelector: #beforeMethod) ir strip.
    after := (self class lookupSelector: #afterMethod) ir strip.

    self replaceSelf: before. self replaceSelf: after.
    self method insertBefore: before startSequence after: after startSequence.

BSMethodWrapper>>replaceSelf: ir  "replace self with pointer to me"
   ^ ir allInstructions do: [:instr | instr isSelf ifTrue: [
                                  instr replaceWith: (IRInstruction pushLiteral: self)]].
```

As we can see, method wrappers are straightforward to implement with BYTESURGEON. The complete implementation included in the distribution consists of 41 lines of code, with comments. This implementation of method wrapper should only serve as an example of use of BYTESURGEON; it is not meant to be a replacement yet since not all features of method wrappers are supported. Furthermore, standard method wrappers and BYTESURGEON method wrappers have different performance profiles.

A Simple Runtime MOP

We now show how to implement a simple runtime MOP for controlling accesses to instance variables. A meta-object can be associated to a class, and upon accesses to instance variables of objects of the class, it gets control via either its instVarRead:in: method (if it is a read access) or its instVarWrite:in:value: method (if it is a write access). For instance, the following TraceMO simply outputs to the transcript what is happening and then performs the standard action, *i.e.*, returning the instance variable value, or storing the new value:

```
TraceMO>>instVarRead: name in: object
    | val |
    val := object instVarNamed: name.
    Transcript show: 'var read: ', val printString; cr.
    ^val.

TraceMO>>instVarStore: name in: object value: newVal
     Transcript show: 'var store: ', newVal printString; cr.
     ^object instVarNamed:  name put: newVal.
```

This meta-object can be installed on class Point as follows:

```
MOP install: TraceMO new on: Point
```

The MOP»install method uses BYTESURGEON to replace the bytecodes that read or store instance variables with calls to the meta-object (*aka.* hooks):

```
MOP class >> install: mop on: aClass
 | dict |
 dict := Dictionary newFrom: {#mo -> mop}.
 aClass instrumentInstVarAccess: [:instr |
    dict at: #name put: instr varname.
    instr isRead
      ifTrue: [instr replace: '<: #mo> instVarRead: <meta: #name> in: self'  using: dict]
      ifFalse: [instr replace: '<: #mo> instVarStore: <:meta #name> in: self
                  value: <meta: #newvalue>' using: dict] ].
```

The dict dictionary is used to hold the reference to the meta-object controlling accesses, and for each access instruction, the name of the variable is put in it. This makes it possible to use user-defined variables when specifying the transformation.

Furthermore, since BYTESURGEON supports runtime bytecode manipulation, we are able to completely *uninstall* hooks when needed:

```
MOP uninstall: MOExample.
```

Of course, this simple MOP is not complete: if methods are changed (recompiled), the MOP is removed, there is no way to compose multiple meta-objects on the same class, it is not possible to associate different meta-objects to different instances, etc. But the basic features are there: a MOP for instance variable accesses that can be installed and retracted at runtime and completely implemented in *less than 10 lines*.

Benchmarks

We now report on several preliminary benchmarks[5] we have performed to evaluate the efficiency of BYTESURGEON. First, we report on transformation vs. compilation costs, and then study the performance of the standard implementation of method wrappers with that based on BYTESURGEON.

Transformation performance. One of the reasons for working on bytecode instead of source is performance. We have carried out a simple set of benchmarks, in which we compare the time to compile some code with both the standard compiler of Squeak and the new compiler (closure compiler), and the time taken by BYTESURGEON to transform all instructions in the code with an empty block. Hence what we actually measure for BYTESURGEON is the time it takes to decompile methods to IR, execute the block for each instruction (which does nothing), generate a new identical method and install it.

The first benchmark is applied to the Object class:

```
"Test compilers"
[Object compileAll] timeToRun

"Test ByteSurgeon"
[Object instrument: [:inst | self ]] timeToRun
```

Class Object contains 461 methods, amounting to 2468 lines of code. We did the same experiment on a larger code base: the whole hierarchy

[5] Machine used: Apple MacBook Pro, 2.4Ghz, Squeak 3.9

	Object		Collections	
	time (ms)	factor	time (ms)	factor
BYTESURGEON	292	1	2264	1
standard compiler	563	1.92	3940	1.74
closure compiler	1924	6.59	16267	7.19

Figure 3.5: Comparing compilation and transformation times.

of collection classes. This hierarchy consists of 79 classes, 2277 methods, summing up to 16122 lines of code. The benchmark is run as:

```
"Test compilers"
[Collection allSubclasses do: [ :c | c compileAll ]] timeToRun

"Test ByteSurgeon"
[Collection allSubclasses do: [ :c | c instrument: [ :inst | self ]]] timeToRun
```

The results of both benchmarks are presented in Figure 3.5. As expected, BYTESURGEON performs very well. The highly optimized standard compiler is approximately twice as slow as BYTESURGEON, while the new compiler, which is much easier to reuse and extend but less optimized, is around 6 times slower. As the new compiler and BYTESURGEON share the same code generator (the IR subsystem), this shows that decompiling from bytecode to IR is indeed faster than compiling source-code to the same IR.

Method wrapper performance. We now compare the performance of the standard implementation of method wrappers with that based on BYTESURGEON. We compare both installation (transformation) time and execution time.

The test consists of a simple before/after counter manipulation wrapping a straightforward method:

```
Bench>>run          beforeMethod          afterMethod
    ^ 3+4.            BCounter inc          BCounter inc
```

The benchmark of the installation/uninstallation is run as follows:

```
[10000 timesRepeat: [
  w := TestMethodWrapper on: #run inClass: Bench.
  w install. w uninstall]] timeToRun
```

The runtime performance of both implementations is compared to that of a method that directly implements the wrapper:

Method Wrapper implementation	Installation time (ms)	Installation factor	Runtime time (ms)	Runtime factor
Hancoded	-	-	10161	1
Standard	1102	1.0	28443	2.8
BYTESURGEON	13835	12.55	10305	1.01

Figure 3.6: Comparing installation and runtime performance of method wrapper implementations.

```
Bench>>run
    BCounter inc.
^   [3+4] ensure: [BCounter inc].
```

The benchmark for both cases is run as follows:

```
GPCounter reset.
b := Bench new.
[10000000 timesRepeat: [b run]] timeToRun
```

The results of the benchmarks (Figure 3.6) show that BYTESURGEON is slower for installing wrappers. This was expected because method wrappers simply swap the wrapped compiled method with the wrapper one, while BYTESURGEON actually modifies the original method. The other side of the coin is that BYTESURGEON-based method wrappers are more efficient at runtime. Standard method wrappers are 2.8 times slower than the hand-coded version, while the BYTESURGEON implementation is as fast as the hand-coded version.

3.2.6 Conclusion

This section has presented BYTESURGEON, a framework for transforming bytecode at runtime realized in Smalltalk. We have shown the power of this framework by realizing method-wrappers and a small MOP. We presented benchmarks to show that the system is practically usable.

The next step now will be to realize *unanticipated partial behavioral reflection* using BYTESURGEON as the underlying mechanism for bytecode manipulation.

3.3 GEPPETTO: Unanticipated Partial Behavioral Reflection

Now that we can insert new code easily at any place in our program *at runtime* using the bytecode transformation framework, we can realize *unanticipated* partial behavioral reflection.

3.3.1 Running Example

Let us consider a collaborative website (a Wiki), implemented using the Seaside web framework [50, 110]. When under high load, the system suffers from a performance problem. Suppose users are reporting unacceptable response times. As providers of the system, our goal is to find the source of this performance problem and then fix it. First, we want to get some knowledge about possible bottlenecks by determining which methods consume the most execution time. A simple profiler shall be applied to our Wiki application, but it is not possible to shutdown the server to install this profiler. During profiling our users should still be able to use the Wiki system as usual. Furthermore, once all the necessary information is gathered, the profiler should be *removed* entirely from the system, again without being forced to halt the Wiki. We have also the *strict* requirement to profile the application in its production environment and context, because unfortunately the performance bottleneck does not seem to occur in a test installation.

To profile method execution we use simple reflective functionalities. We just need to know the name and arguments of the method being executed, the time when this execution started and the time when it finished to gather statistical data, showing which methods consume the most execution time. During the analysis of the execution time of the different methods we see that some very slow methods can be optimized by using a simple caching mechanism. We then decide to dynamically introduce a cache for these expensive calculations in order to solve our performance problem.

As we see in this simple but realistic example, the ability to use reflection is of wide interest for systems that cannot be halted but nonetheless require reflective behavior temporarily or permanently. Furthermore, this example proves that an approach to reflection has to fulfill two important requirements to be applicable in such a situation: first, the reflective architecture has to allow *unanticipated installation and removal* of reflective behavior into an application at runtime. A web application or any other server-based application can often not be stopped and restarted to install new functionality. Moreover, the use of reflection cannot be anticipated

before the application is started, hence a preparation of the application to support the reflective behavior that we may want to use later is not a valid alternative here. So the reflective mechanisms have to be inserted in an unanticipated manner. Second, in order to be able to use reflection in a durable manner (*e.g.*, for caching) in a real-world situation, the reflective architecture has to be efficient. This motivates the need for *partial* reflection allowing the programmer to precisely choose the places where reflection is really required and hence minimizing the costs for reflection by reducing the amount of costly reifications occurring at runtime. To sum up, this example requires *unanticipated partial behavioral reflection* to be solved.

Partial Behavioral Reflection in a Nutshell

GEPPETTO adopts the model of partial behavioral reflection (PBR) presented in [128], which we hereby briefly summarize. This model consists of explicit *links* binding *hooksets* to *meta-objects* (Figure 3.7).

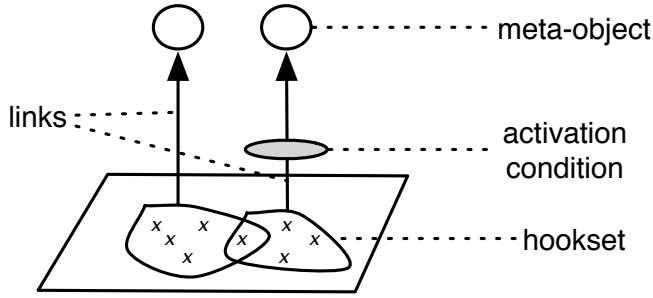

Figure 3.7: Links are explicit entities bindings hooksets (at the base level) to meta-objects, possibly subject to activation conditions.

A *hookset* identifies a set of related operation occurrences of interest, at the base level. A *meta-object* is a standard object that is delegated control over a partial reification of an operation occurrence at runtime. A *link* specifies the causal connection between a hookset (base level) and a meta-object (meta-level). When occurrences of operations are matched by its hookset, the link invokes a method on the associated meta-object, passing it pieces of reified information. Exactly which method is called, and which pieces of information are passed, is specified in the link itself. So, the link specifies the expected meta-object protocol, and the meta-object can be any object fulfilling this protocol.

Several other attributes further characterize a link, such as the *control*

that is given to the meta-object (*i.e.*, that of acting before, after, or around the intercepted operation occurrence). A dynamically-evaluated *activation condition* can also be attached to the link, in order to determine if a link applies or not depending on any dynamically-computable criteria (*e.g.*, the amount of free memory or the precise class of the currently-executing object).

As mentioned earlier, PBR achieves two main goals: *(1)* highly-selective reification, both spatial (which occurrences of which operation) and temporal (thanks to activation conditions), and *(2)* flexible meta-level engineering thanks to fine-grained protocol specification and the fact that a hookset can gather heterogeneous execution points (*i.e.*, occurrences of different operations in different entities).

The following short example illustrates the above definitions. Recall the slow collaborative website mentioned in section 3.3.1. To profile this application we dynamically introduce a profiler with GEPPETTO, analyzing the method #toughWork which we suspect of being responsible for the performance issues.

First, we select this method by defining a hookset. This hookset also selects the operation to be reified, in this case the evaluation of the method #toughWork:

```
toughWorks := Hookset inClass: 'WikiCore' inMethod: #toughWork.
toughWorks operation: MethodEval.
```

Second, we specify the link which bridges the gap between the base level (*i.e.*, method #toughWork) and the meta-level (*i.e.*, the meta-object, an instance of class Profiler). The link also describes the call to the meta-object, *i.e.*, which method to invoke on the meta-object, specified by passing a meta-level selector.

```
profiler := Link id: #profiler hookset: toughWorks metaobject: Profiler new.
profiler control: Control around.
profiler metalevelSelector: #profile:.
```

After having installed this link by executing profiler install the method #profile: of the meta-object will be executed on every call to method #toughWork of class WikiCore. The developer can provide an arbitrarily complex implementation of the profiler meta-object. The next section shows a more elaborated version of this profiling example.

3.3.2 Solving the Running Example with GEPPETTO

To illustrate the use of GEPPETTO, we now explain how to solve the problem introduced in Section 3.3.1. In order to find out where the performance issue comes from, we start by elaborating a meta-object protocol to profile the Wiki application. Once we have identified the costly methods that can be cached, we introduce a caching mechanism with GEPPETTO.

Profiling MOP

Defining and introducing dynamically reflective behavior into an application consists of three steps: The first step is the specification of the places where metabehavior is required (*e.g.*, in which classes and methods, for which objects) by configuring a hookset. In the second step the definition of the meta-object protocol (*e.g.*, which data is passed to which meta-object) is specified by setting up one or more links. Third and finally, we perform the installation of the defined reflective functionality.

For profiling method execution of our Wiki application, we need to define a link, binding the appropriate hookset to a Profiler meta-object. The hookset consists of all method evalution occurrences in all classes of the Wiki application. Hence the hookset is defined as follows:

```
allExecs := Hookset new.
allExecs inPackage: 'Wiki'; operation: MethodEval.
```

All classes of the Wiki package are of interest, and any occurrences of a method evaluation as well.

Now we have to specify which method of the meta-object has to be called, and when. In order to be able to determine the execution time of a method, the profiler acts *around* method evaluations, recording the time at which execution starts and ends, and computing the execution time. The link, called profiler, knows the meta-object to invoke, an instance of class Profiler:

```
profiler := Link id: #profiler hookset: allExecs metaobject: Profiler new.
profiler control: Control around.
```

The profiler therefore needs to receive as parameters the selector being sent, the class to which the method being evaluated belongs and the arguments. The method to call on the profiler object is thus profileMethod:inClass:withArguments:. This protocol is described by sending the following message to the profile link:

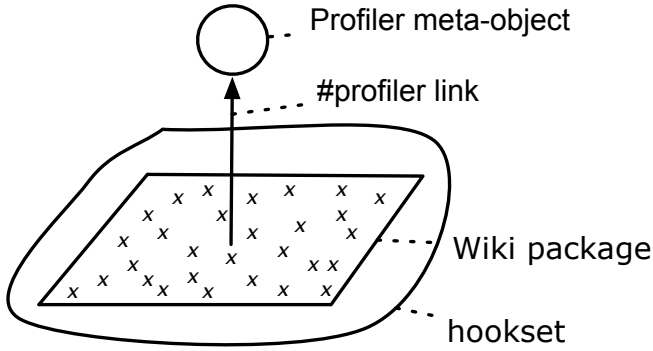

Figure 3.8: The profiler hookset affects the whole Wiki application.

```
profiler metalevelSelector: #profileMethod:inClass:withArguments:
    parameters: {Parameter selector. Parameter methodClass. Parameter
    arguments}
    passingMode: PassingMode plain.
```

The class Parameter is used to describe exactly which information should be reified and how to pass it to the meta-level. See Section 3.3.3 for more information.

Profiler is a conventional Smalltalk class, whose instances are in charge of handling the task of profiling. For the sake of conciseness, we do not explain the implementation of such a profiler. Finally, to effectively install the link, we just need to execute:

```
profiler install.
```

and GEPPETTO inserts all required hooks. From now on, all method executions in the Wiki application get reified and the Profiler meta-object starts gathering data.

To better understand how the installed meta behavior changes the execution of the Wiki application we present a sequence diagram depicting the execution flow on the basis of a small example. This diagram shows how the control flows from the main method (this code example) over the base level (the Wiki application code) to the meta-level (the profiler object).

```
"editing a page"
page := WikiModel at: pageName.
page title: newTitle.
```

```
doc := DocumentParser parse: wikiText.
page document: doc.
```

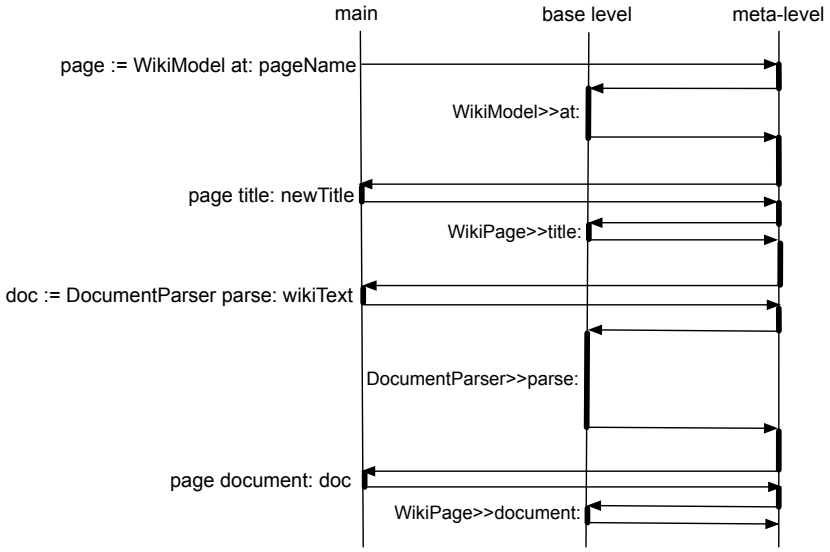

Figure 3.9: Execution flow in the Wiki during the editing of a page when the profiler is installed

Now suppose that based on the gathered data, we determine that a particular method is indeed taking much time: #visitPage: of our Wiki Visitor objects. This method is responsible for building up recursively all the HTML code of a wiki page. It fortunately happens that this method can seemingly benefit from a simple caching mechanism. We can now completely remove the profiling functionality from the Wiki, reverting to normal execution, without any reification occurring anymore. This is achieved simply by executing:

```
profiler uninstall.
```

GEPPETTO then dynamically removes all hooks from the application code, hence further execution is not subject to any performance overhead.

As we use BYTESURGEON to do the actual modification to the system (see Section 3.3.4), all change is done at the granularity of a single method. All existing, running executions continue to execute the old code, only new invocations are using the changed code after the method is installed. The

introduction of links thus is atomic on a per-method basis. The clients of the framework have to take this into account. Code executing in a concurrent thread to the one setting the links will not see all links of the hookset introduced in one atomic step, but one-by-one at method granularity. In practice, this has not yet lead to problems. Nevertheless, advanced control of which part of the system is affected by such reflective change and the ability to do large scale change in one atomic, transactional step is an interesting question for future research.

Caching MOP

We now explain how the caching functionality is dynamically added with GEPPETTO. First, we define the hookset and then the link:

```
toughWorks := Hookset new.
toughWorks inClass: Structure inMethod: #visitPage: operation: MethodEval.

cache := Link id: #cache hookset: toughWorks metaobject: Cache new.
cache control: Control around.
cache metalevelSelector: #cacheFor:
    parameters: {Parameter arg1}
    passingMode: PassingMode plain.
```

The only piece of information that is reified is the first argument passed to the #visitPage: method, which is the page being visited, denoted with Parameter arg1.

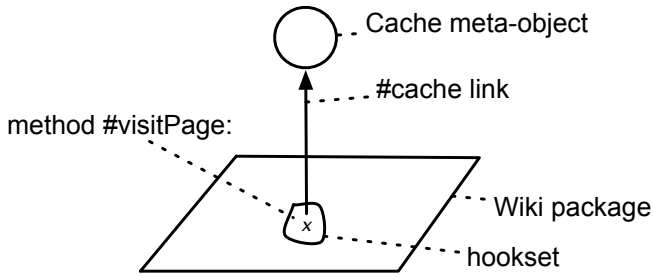

Figure 3.10: The cache hookset only affects method #visitPage:.

Cache is a Smalltalk class whose instances manage caching (based on the single parameter value). In the #cacheFor: method, we first check if the cache contains a value for the passed argument. If so, this value is returned by the meta-object. Else, the meta-object proceeds with the replaced operation

Page	w/ cache (ms)	w/o cache (ms)	Optimization
Page 1	29	2758	95x
Page 2	35	8529	244x
Page 3	32	2461	77x

Table 3.1: Effect of the caching meta behavior for rendering one thousand times the HTML code of three wiki pages.

of the base level, takes the result answered by this operation via #proceed and returns this value after having stored it into the cache:

```
cacheFor: aPage
  | result |
  (self cacheContains: aPage) ifTrue: [^self cacheAt: aPage].
  result := self proceed.
  self cacheAt: aPage put: result.
  ^result
```

In order to be able the to proceed with the original operation the class of the meta-object has to inherit from the generic class ProceedMO. All instances of subclasses of ProceedMO are allowed to proceed with replaced operations.

Installing the cache is simply done by executing cache install. GEPPETTO inserts the necessary hooks in the code, and from then on, all evaluations of the #visitPage: method are optimized by caching.

The effect of this cache is tremendous. We compare the situation with and without this cache installed by generating one thousand times the HTML code of three exemplary wiki pages with complex content such as links and tables.[6] Without the active caching mechanism the HTML code is completely built up on every single visit to a wiki page, whereas otherwise the HTML code is taken from the cache. As Table 3.1 shows we achieve an average optimization of almost factor 140 with an installed cache for the result of method #visitPage:.

Although this example is pretty straightforward, it illustrates well the point of UPBR: one can easily add reflective features at runtime, with the possibility to completely remove them at any time. This fosters incremental and prototypical resolution of problems such as the one we have illustrated. For instance, if it turns out that the introduced caching is not effective enough, it can be uninstalled, and a more elaborate caching can be devised.

[6]These benchmarks were executed on a MacOS X server with an Intel Core 2 Duo 2.16 GHz processor and 1 GB of RAM.

Persistence MOP

An another issue of our Wiki application is the persistent storage of its data. Currently, all the data is only stored in the Smalltalk image which is not really fail-safe: we might lose data when the image crashes. Hence we want to store the Wiki data persistently in a relational database. To quickly evaluate if it is possible to use a relational database without having to change the code, we implement an experimental storage mechanism using GEPPETTO. This persistence mechanism works simply by transforming every store access to an instance variable such as title or text in class Page to write-through the variable's value into a database. On every read access to such an instance variable we access transparently the same database to get the value for the variable from there.

We can easily select every store access to an instance variable in class Page with the following hookset:

```
storeHookset := Hookset new.
storeHookset inClass: Page; operation: InstVarAccess.
storeHookset operationSelector: [:varAccess | varAccess isInstVarStore].
```

We simply specify a hookset affecting the whole Page (*e.g.*, every method of it) and selecting every instance variable access which is a store (as defined with the operation selector).

Next we define a link taking the above hookset and specifying the meta-object and the invocation of it:

```
storeLink := Link id: #storePersistence hookset: storeHookset
                metaobject: DBPersistenceMO new.
storeLink control: Control after.
storeLink metalevelSelector: #storeInstVar:withValue:of:
        parameters: {Parameter varName. Parameter varNewValue. Parameter self}
        passingMode: PassingMode plain.
```

We provide a class DBPersistenceMO holding the required behavior to actually store the content of an instance variable into the database. The method #storeInstVar:withValue:of: needs to know the name of the instance variable, the value which is stored into, and the instance of Page in which the store occurs. With this data, the meta-object is able to store the whole content of a Wiki page transparently into a database. The code to actually store the content of an instance variable into a database can be arbitrarily complex. The meta-object is invoked after the original instance variable store takes place which means that the Page object has already the correct value stored in its instance variable when the value is written to the database.

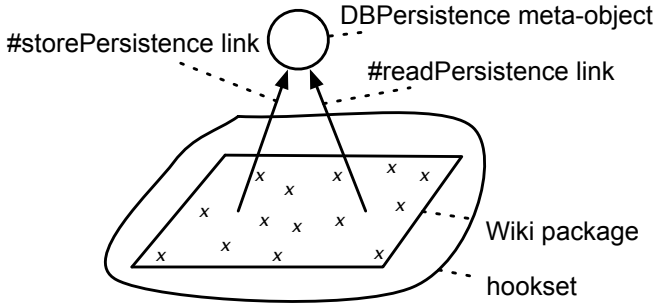

Figure 3.11: The persistence hookset encloses the whole Wiki application, but only affects methods containing instance variable accesses (read or write).

To complete this example we also present the inverse of storing instance variables into the database, namely fetching the content of an instance variable directly from the database on every read access.

First, we give the hookset definition:

```
readHookset := Hookset new.
readHookset inClass: Page; operation: InstVarAccess.
readHookset operationSelector: [:varAccess | varAccess isInstVarRead].
```

The only difference to the store version of this hookset is that we now check for an instance variable read in the operation selector.

Second, we give the definition of the link between the readHookset and the meta-object fetching the value of an instance variable out of a database:

```
readLink := Link id: #readPersistence hookset: readHookset
            metaobject: DBPersistenceMO new.
readLink control: Control around.
readLink metalevelSelector: #readInstVar:of:
        parameters: {Parameter varName. Parameter self }
        passingMode: PassingMode plain.
```

As the inverse of the store link, this read link acts around the original instance variable read access: it replaces the read access entirely and lets the meta-object insert the value of the instance variable. The meta-object is still an instance of class DBPersistenceMO, but this time the method #readInstVar:of: is invoked, expecting the name of the instance variable and the Page object as parameters. This method will query the database for the correct value of the given instance variable.

This very simple persistence mechanism can already provide us with information about the efficiency and accuracy of using a relational database as a backend for our Wiki application. GEPPETTO also allows the programmer to easily experiment with other persistence mechanisms, *e.g.*, techniques based on XML.

3.3.3 GEPPETTO Design

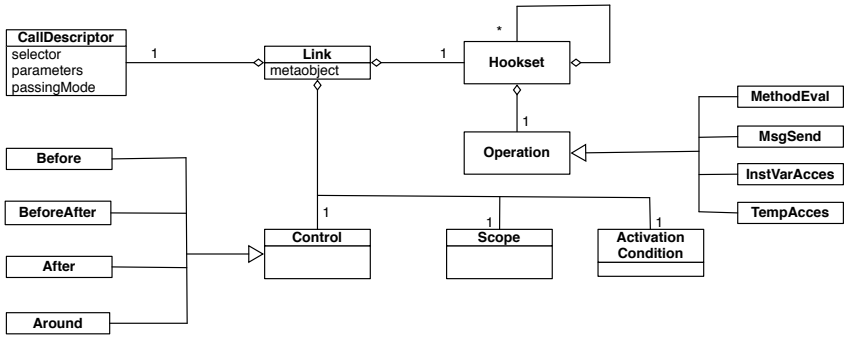

Figure 3.12: Class diagram of GEPPETTO design

GEPPETTO instantiates the model of partial behavioral reflection previously presented, as summarized on Figure 3.12. A link binds a hookset to a meta-object, and is characterized by several attributes. A hookset specifies the operation it matches occurrences of, which can be either MethodEval, MsgSend, InstVarAccess or TempAccess. Hooksets can also be composed as will be explained later.

Spatial selection of operation occurrences in GEPPETTO can be done in a number of ways, as shown in Table 3.2. Eventually, occurrences are selected within method bodies (or boundaries), by applying an *operation selector*, *i.e.*, a predicate that can programmatically determine whether a particular occurrence is of interest or not. Coarser levels of selection are provided to speedup the selection process. First of all, one can eagerly specify the operation of which occurrences may be of interest. Furthermore, one can restrict a hookset to a given package, to a set of classes (using a *class selector*), or to a set of methods (using a *method selector*). Convenience methods are provided when an enumerative style of specification is preferred.

To select for instance every class in the system whose name contains the string 'Wiki' we use this expression:

```
hookset classSelector: [:class | class name includesSubString: 'Wiki'].
```

Selection Level	Example
Package	hookset inPackage: 'Wiki'
Class	hookset classSelector: [:class | class superclass = MyClass] hookset inClasses: { MyClass. YourClass}
Method	hookset methodSelector: [:meth | meth selector = #hello] hookset inMethods: { #hello. #bye}
Operation	hookset operation: MsgSend
Occurrence	hookset operationSelector: [:send | send selector = #size]

Table 3.2: Spatial Selection in GEPPETTO

A class selector is evaluated for every class existing in the system, a method selector is evaluated for all methods every selected class provides. If the above class selector selects the classes WikiPage and WikiFolder then the following method selector is evaluated for all methods in WikiPage as well as for all methods in WikiFolder:

```
hookset methodSelector: [:meth | meth selector = #content].
```

To enumerate the desired classes and methods directly instead of defining a to be evaluated predicate, one can simply pass an array of classes or methods:

```
hookset classes: {WikiPage. WikiFolder}.
hookset methods: {#content}.
```

Often it is much easier to enumerate the desired entities directly than coming up with selectors.

Thus far, hooksets are operation-specific. Like in Reflex, GEPPETTO supports hookset composition, so a hookset can match occurrences of different operations. Hooksets can be composed using union, intersection, and difference.

To get a hookset which is the union of two single hooksets, we write:

```
unionHookset := CombinedHookset union: hookset1 with: hookset2.
```

This unionHookset selects all operation occurrences that hookset1 and hookset2 together select. The other set operations are implemented in methods called #intersection:with: and #differenceBetween:and: on the class side of CombinedHookset.

If some hooks of different hooksets conflict with each other, *e.g.*, more than one hookset affects a particular occurrence of a message send in a

given method, then these hooks are automatically composed by GEPPETTO. In a composed hook every single hook is executed in sequence in the order of their installation time.

See section 3.3.4 for details about hook composition.

A `Link` object is created by giving an identifier, the hookset, and by specifying how the meta-object instance(s) are to be obtained.

```
link := Link id: #profiler hookset: hs metaobjectCreator: [ Profiler new ]
```

The block given for the meta-object creator is evaluated to bootstrap meta-object references. As a shortcut, one can directly give a meta-object instance, instead of a block; the given instance will then be shared among entities affected by the link.

A link is further characterized by several attributes:

- Control defines when the meta-object associated to the link is given control over an operation occurrence: it can be either Before, After, BeforeAfter or Around. BeforeAfter means that the meta-object is called *before* and *after* the original operation, whereas Around replaces the operation. The replaced operation then can be executed by calling proceed in the meta-object, if this meta-object is an instance of a subclass of ProceedMO.

- Scope determines the association scheme of a meta-object with respect to base entities. For instance, if the link has object scope, then each instance affected by the link has a dedicated meta-object for that link. The scope can also be *class* (one meta-object per class), or *global* (a unique meta-object for the link).

- an ActivationCondition is a dynamically-evaluated predicate that determines if a link is active (that is, whether reification and delegation to the meta-object effectively occurs). A typical usage of an activation condition is to obtain object-level reifications: the condition can be used as a discriminator of instances that are affected or not by the considered link. The predicate defining the activation condition receives the current object (*i.e.*, the object in which the hook is executed) as its sole parameter.

- a CallDescriptor defines the communication protocol with the meta-object. A call descriptor embeds the selector of the message to be sent, the parameters to pass as well as *how* they are passed (*i.e.*, as plain method arguments, packed into an array, or embedded in a wrapper object). Table 3.3 lists all possible parameters depending on the reified operation.

Operation	Reified Data	Description
All Operations	context	execution context
	self	the object
	control	before, after or replace
Message Send/ Method Evaluation	arguments	arguments as an array
	argX	X^{th} argument
	sender	sender object
	senderSelector	sender selector
	receiver	receiver object
	selector	selector of method
	result	returned result (after only)
Temp/InstVar Access	name	name of variable
	offset	offset of variable
	value	value of variable
	newvalue	new value (write only)

Table 3.3: Supported reified information

Meta-objects have to be set differently depending on the *scope* attribute of the link. The convenient methods mentioned above, #metaobject: and #metaobjectCreator:, are valid for global scope where the whole link has either one meta-object, or every reflective object has its own meta-object instance. But one can also precisely define which reflective object should have which meta-object when using object scope. The method #setMetaobject:forObject: lets us specify which meta-object is valid for which reflective object. Similarly one can use method #setMetaobject:forClass: to associate dedicated meta-objects with reflective classes when using class scope.

To specify a dynamically evaluated activation condition we can either pass a block holding this condition or implement a subclass of Active. For complex activation conditions it is recommendable to implement a dedicated class which also enhances the possibilities to reuse the defined condition later on. To implement such a class-based activation condition, we just need to override #evaluate: of Active. This method expects as a parameter the current object in which the hook is being executed. To execute a hook only if it occurs in a certain object, (*i.e.*, to obtain object-level reification) we provide a very simple implementation of #evaluate::

```
ObjectLevelActive >> evaluate: anObject
  ^anObject = self predefinedObject.
```

With the following code we inform the link to use this activation condi-

tion:

```
link active: (ObjectLevelActive object: predefinedObject).
```

To get the same activation predicate using a block we simply write:

```
link activationCondition: [:object | object = predefinedObject].
```

The link gets asked by the hook if it is active or not. The link itself asks the associated activation condition if it evaluates to *true* for the given object. If so, the hook is further executed to reify the necessary data and to finally invoke the meta-object. Otherwise, the hook immediately gives the execution to the next operation.

To use the call descriptor one can create explicitly an instance of class CallDescriptor:

```
callDesc := CallDescriptor selector: #msgSend:
                    parameters: {Parameter arguments. Parameter receiver}
                    passingMode: PassingMode array
```

The call descriptor defines that an array containing the arguments and the receiver of a message send has to be passed to the method #msgSend: of the meta-object. We install this call descriptor by invoking the link method #callDescriptor: and passing the call descriptor object to it.

The link also provides convenience methods to implicitly create the call descriptor. The following code is equivalent to the above:

```
link metalevelSelector: #msgSend:
   parameters: {Parameter arguments. Parameter receiver}
   passingMode: PassingMode array
```

Finally, for a link to be effective, it has to be dynamically installed by sending the install message to it. At any time, a link can be uninstalled via uninstall. Links have identifiers, which can be used to retrieve them from a global repository at any time (Link get: #linkID).

3.3.4 Implementation Issues

In this section we explain a crucial part of the implementation of GEPPETTO: the installation of hooks in the bytecode. As explained earlier, we have to dynamically install hooks at runtime to be able to apply reflection in an unanticipated manner to a running system. Therefore, we require a means to manipulate bytecode at runtime. For that purpose we use BYTESURGEON, the framework for runtime manipulation of bytecode presented

in Section 3.2. Using this tool we do not have to work directly with bytecode. Instead we write our hooks in normal Smalltalk code, which we then pass to BYTESURGEON. Internally, BYTESURGEON will compile our code to bytecode and insert the resulting bytecode into compiled methods.

Adapting Method Bytecode

To adapt the bytecode of methods, we first select the method in which we want to change the bytecode (recall that a method is defined as the combination of a class and a selector, *e.g.*,`WikiPage#document`). Second, we instrument this method with one of the instrumentation methods added by BYTESURGEON to compiled methods, *e.g.*,`#instrumentSends:` or `#instrumentInstVars:`, to access all the specific operations in a method, *i.e.*, message sends or instance variables accesses, respectively. These instrumentation methods expect a block as single argument. In this block we have access to a block argument which denotes the current operation occurrence object. For a message send we get access to an instance of `IRSend` (this is part of the intermediate representation on which BYTESURGEON is based, see Section 3.2.3).

Below is a short example showing how BYTESURGEON can be used to insert a simple piece of Smalltalk code into the method `#document` of class `WikiPage`:

```
(WikiPage>>#document) instrumentSends: [:send |
        send selector = #size ifTrue: [ send replace: '7']]
```

In this example we replace every send of the `#size` message occurring in the method `#document` of class `WikiPage` to simply return the constant 7. This example shows how to access different operations in a method (operation selection, *i.e.*, message sending) and how to select different operation occurrences (intra-operation selection; *i.e.*, message sends invoking `#size`) in a method.

During the instrumentation of a method the defined block is evaluated for every such operation in that method. To do intra-operation selection it is enough to specify a condition in the block, such as asking if the selector of an `IRSend` is of interest. Only if this condition is met the corresponding operation occurrence is adapted, either by replacing it or by inserting code before or after it. The code to be inserted is written as normal Smalltalk code directly in a string. In this string we can refer to dynamic information by using *meta-variables*, such as <meta: #$receiver$> or <meta: #$arguments$> to reference respectively the receiver or the arguments of a method.

Structure of a Hook

In GEPPETTO, hooks are inserted in bytecode to provoke reification and delegation at runtime, where and when needed. The execution of a hook is a three-step process:

- It checks if the link is active for the currently-executing object;
- It reifies dynamic information and packs this information as specified by the call descriptor of the link;
- It performs the actual delegation to the meta-object, by sending the message specified in the call descriptor, with the corresponding reified information.

When a link has to be installed, GEPPETTO evaluates the static selectors (package, class, method, etc.) and then generates an appropriate string of Smalltalk code based on the specification of the call descriptor of the link. This string is then compiled and inserted by BYTESURGEON. For instance, for the cache link of Section 3.3.2, the generated Smalltalk code is:

```
(<meta: #link> isActiveFor: self)
  ifTrue: [<meta: #link> metaobject cacheFor: <meta: #arg1>].
```

First, the activation condition is checked. Note that the link itself is available as a meta-variable for BYTESURGEON. If the link is active for the currently-executing object, then second delegation occurs: the meta-object is retrieved from the link, and the #cacheFor: message is sent with the first argument as parameter. Step two and three, reifying dynamic information and performing the delegation to the meta-object, occurs in one and the same line of code by defining a message send whose arguments are the reified information and whose receiver is the meta-object.

The exact string generated depends on the call descriptor defining the message name, parameters, and passing mode. For instance, if the passing mode is by array, it is necessary to first build up the array explicitly in the hook. The generated code also depends on the scope of the link (*e.g.*, if the link has object scope, then retrieving the meta-object requires passing the currently-executing object).

The following code denotes the hook code to send a method to the meta-object when using object scope and array passing mode:

```
(<meta: #link> metaobjectForObject: self) cacheMsgSend:
    (Array with: self with: <meta: #selector>
        with: <meta: #receiver> with: <meta: #arguments>)
```

To cache a message send with a dedicated meta-object for every base level object in which this message send occurs, we opt for object scope. The hook hence asks the link for the meta-object associated with the current executing object. The meta-level message is then sent to the obtained meta-object. The single argument expected by this message is an array which is explicitly built up in the hook. To access the different reifications required, *e.g.*, selector, receiver and arguments, we have again used the meta-variables of BYTESURGEON.

Note that we optimized the look up of the meta-object by storing it automatically into an instance variable for the current executing object when using object scope. Subsequent executions of the same hook or of another hook occurring in the same object can then simply read the meta-object from this instance variable which avoids costly look ups of meta-objects in a dictionary. Meta-objects are only valid for one single link, hence these meta-object instance variables are specific to a certain link to make sure that more than one link can affect a given base level object. A similar optimizing mechanism also exists for class scope where meta-objects are not stored in instance variables, but in class variables.

The complete hook for the more complex cache example above has the following structure:

```
(<meta: #link> isActiveFor: self) ifTrue: [
  (<meta: #link> metaobjectForObject: self) cacheMsgSend:
        (Array with: self with: <meta: #selector>
              with: <meta: #receiver> with: <meta: #arguments>)]
  ifFalse: [<meta: #proceed> value]
```

If the link is not active for the current executing object the original operation has to be executed as denoted in the *false* predicate. The proceed statement continues the execution of the original operation around which the installed hook acts.

The proceed statement provided by BYTESURGEON is also used for the resumable meta-objects presented in Section 3.3.2. To be able to proceed with the original operation in the meta-object GEPPETTO passes the value of the proceed statement (*e.g.*, a message send or an instance variable access) to the meta-object. This proceed value is stored in the instance variable proceed of the meta-object. By sending the message #proceed to a resumable meta-object, a subclass of ProceedMO, this proceed value is evaluated and the execution of the original operation is triggered, *i.e.*, proceeded. Note that only meta-objects that act around a base level operation can proceed with the original operation.

Hook Composition

If more than one hookset is installed in a given application, some hooks of different hooksets may conflict with each other, for instance if two hooksets affect the same message send of a given method. GEPPETTO is capable of detecting and resolving such a conflict automatically at runtime during the installation of every new link.

Detecting a hook conflict is a two-step process: First, GEPPETTO determines for every link that is being installed, if another link also manipulates a given method, *i.e.*, if meta-level behavior is already installed in this method. GEPPETTO holds a global repository containing all installed links with a list of the affected classes and methods for each link. Querying this repository results in a collection of links affecting a given method. Second, GEPPETTO analyzes every instruction of a method to find out where exactly in the method body more than one link does install a hook. Concretely, the hook installer iterates over every instruction of such a method and tests for every conflicting link if it manipulates the current instruction. The following code illustrates this:

```
conflictingLinks do: [:eachLink |
  (method ir allInstructionsMatching: eachLink hookset operationSelector) do: [:instr |
    "this instruction is manipulated by the given link"
    self addLinkToRepository: eachLink forInstr: instr.
].
```

As soon as the hook installer has detected all the instructions conflicting with already installed links as described above, it solves the conflict by collecting first all the hooks manipulating a given instruction. Second, all these collected hooks are installed in sequence before, after or instead of the original instruction, depending on the control attribute specified in the link. The order in the sequence is determined by the installation time of the conflicting links, the first installed link will be installed first.

Note that there is not always a conflict when two links manipulate the same instruction of a method. If one link *e.g.*, executes meta-level behavior before the original instruction and the second one afterwards then these links do not conflict at this instruction. Hence the conflict detection algorithm has to take into account the controls of the links.

Finally, note that GEPPETTO adopts a simple automatic composition strategy; future work may include considering more advanced link composition strategies as supported by Reflex [124]. For example, instead of relying on the order of installation, we want to be able to define the order of composition explicitly.

System	slowdown factor
Geppetto	10.85
Iguana/J	24
MetaclassTalk	20

Table 3.4: Slowdowns of different reflective systems for the reification of message sends.

3.3.5 Evaluation

We now report on preliminary micro-benchmarks that validate the performance of GEPPETTO by comparing it with other reflective frameworks and architectures. Subsequently we conduct a more complex benchmark measuring the efficiency of the profiler we presented in Section 3.3.2 by comparing the execution of some test suites of the Wiki application with and without the profiler being installed in the Wiki.

Micro-Benchmarks

For the first micro-benchmark we measure the slowdown of a fully reified message send over a non-reified message send. In Table 3.4 we compare the reflective systems Iguana/J [108], and MetaclassTalk [18] to GEPPETTO. The measurement for Iguana/J was taken from the paper on Iguana/J [108]. For MetaclassTalk and GEPPETTO, we performed the benchmarks on a Windows PC with an Intel Pentium 4 CPU 3.4 GHz and 3 GB RAM. The version of MetaclassTalk used was v0.3beta, GEPPETTO was running in Squeak 3.9. The master's thesis on GEPPETTO [115] contains more detailed explanation and the source code of the benchmark.

We are comparing systems to GEPPETTO that do not provide partial reflection. As previously mentioned, the real performance gain of partial reflection arises from the fact that we are able to exactly control what to reify and thus are able to minimize the reification costs. This benchmark does not cover this use but lets GEPPETTO reify every information about a message send to be comparable with the other systems. The benchmark will thus only give an impression of the worst case, *i.e.*, when GEPPETTO is doing full reification of a message send.

Because Iguana/J uses Java, we cannot directly compare its execution times with those of GEPPETTO. So we performed such a comparison with MetaclassTalk, since both GEPPETTO and MetaclassTalk are running in the same environment. We implemented for the operations message sending

	MetaclassTalk (ms)	GEPPETTO (ms)	Speedup
message send	108	46	2.3x
instance variable read	272	92	2.9x

Table 3.5: Speedup of GEPPETTO over MetaclassTalk for reified message send and instance variable read access.

and instance variable access the same meta-object protocol and the same behavior at the meta-level in both proposals to be able to compare the resulting execution time. The measured execution time includes the reification as well as the processing of the meta-level behavior. For message sending we reify the receiver, the selector and the arguments, for instance variable access the name of the variable and its value. Table 3.5 presents the results of the benchmark. The Windows PC mentioned above was also used to execute this benchmark. For both operations, message send and instance variable access, we reified almost every possible information in GEPPETTO to get a reliable comparison with MetaclassTalk which does not support controlling which information should be reified. Hence GEPPETTO will perform even better than the 2-to-3 times speedup compared with MetaclassTalk in cases where not all information about an operation occurrence is required.

The reason why GEPPETTO is so much faster than MetaclassTalk lies in the approach to message reification. MetaclassTalk wraps every method (using MethodWrappers [24]) by default to allow all message receives to be reified even when called from a class not under the control of MetaclassTalk. GEPPETTO on the other hand does not try to provide reified message reception in this case, as we requested only a reification of message sending.

Benchmarking the Profiler

We conducted a third benchmark measuring the general slowdown caused by reflective behavior introduced with GEPPETTO. We go back to the Wiki example of Section 3.3.2 where we installed a profiler in this application and compare now the execution times of the Wiki test suite with and without the profiler being active. The test suite contains 131 unit tests. The profiler itself acts around every method evaluation in the Wiki parts Structure as well as Visitor and measures the time required to execute the methods in these packages by stopping the time before and after the execution of the methods. The execution of the original method is triggered in the profiler meta-object by using the proceed statement explained in Section 3.3.2. The obtained execution time is then stored in a dictionary with the profiled

Test suite	# tests	w/ profiler (ms)	w/o profiler (ms)	Slowdown
Structure tests	111	661	76	8.7x
Decoration tests	20	23	8	2.9x

Table 3.6: Overall slowdown caused by the profiler in the meta-level

methods as a key and a collection of measured times as value. As discussed in Section 3.3.2 we only reify the selector, the arguments and the class to which the method belongs and pass this information in plain mode to the profiler.

Table 3.6 contains the results obtained by running the benchmark on the Wiki server also used in Section 3.3.2 to measure the efficiency of the caching meta behavior. Clearly, the active profiler causes a slowdown between factor 3 and 8. Further benchmarks show that more than 50% of this slowdown is caused by the execution of the profiling code itself, which means that the reification and the invocation of the meta-object is not as much responsible for the high costs of this meta-level profiler as the profiler implementation itself.

These benchmarks indicate that the applied model for partial behavioral reflection is efficient compared to other models. Hence the combination of PBR and UBR is indeed fruitful and successful, because UPBR enables us to use unanticipated reflection in an efficient and effective manner.

3.4 Problems of the Approach

The work shown in this chapter focused on solving just one of the problems noted for behavioral reflection: unanticipated use. We successfully realized a system that provides unanticipated partial behavioral reflection in a very practical manner. We have shown a number of examples and presented benchmarks.

But not everything is perfect. The use of bytecode transformation proves to be problematic.

Semantic mismatch. There is first the problem that not all concepts present in the language have a suitable representation on the level of bytecode.

- *Variables* in bytecode. We have operation for variable read and write. But on the level of the bytecode, we only distinguish between temporary, instance and literal variables. Literal variables represent class, pool and global variables.

- *Block Closures* do not have a direct static counterpart in the bytecode. Instead, there is code that generates blocks at runtime, making it hard to reflect on them statically.

- *Control Structures* are optimized to jumps on the level of the bytecode. Even though the user thinks about ifTrue: as a message send, it it not a message send on the level of the bytecode and can not be reflected upon using our framework.

Problem with synthesized elements. Since we use bytecode as our sole representation, there is a problem: we destroy this representation when inserting the code to call the meta-objects. After using behavioral reflection, the structure is changed and it is not possible to distinguish original code from added code for the links.

There is a possible solution for this: we could build up elaborate structures to remember which bytecode offset is original and which is generated, but this will get soon very complicated in practice. The real problem is that the representation used for structural reflection should not be modified to realize behavioral reflection. We will see that this problem can be solved with the help of annotations on a suitable sub-method structural model in Chapter 5.

Code quality problem due to low level model. We have seen in the section on ByteSurgeon that the access to runtime information like arguments and receiver of message sends, needs quite some effort when working on bytcode.

Bytecode manipulation is always a local change, at the point of a message send, we just know that the stack has a certain layout. This information we can use: we need to store information from the stack in local variables, then we can use them. Afterwards we have to rebuild the stack again.

These problems are related to the fact that we do not use the right representation of sub-method structure, with a higher level representation we can generate optimized code.

3.5 Conclusion and Summary

We have shown how to realize unanticipated partial behavioral reflection. We first described ByteSurgeon, a bytecode transformation framework for Smalltalk. We then discussed how to realize partial behavioral reflection and presented case-studies for its use. Benchmarks validate the practicability of our approach.

Nevertheless, there are problems that lead us to abandon this bytecode-transformation based approach. The next chapter will present *sub-method* structural reflection which provides a better basis for realizing partial behavioral reflection.

Chapter 4

Sub-Method Reflection

4.1 Introduction

Now we describe sub-method structural reflection. We analyze the problems of finding a suitable model for structural reflection that does not stop at the method level. We describe how to realize this model in a normal bytecode-based environment and present examples of systems realized using sub-method reflection. We discuss performance and memory consumption. The chapter closes with an overview of related work.

4.2 Challenges for Supporting Sub-Method Reflection

To support the implementation of different tools such as a code browser, a code coverage tool or a refactoring engine, we need a representation that is extensible, it should allow tools to annotate the structural elements with metadata and the use of metadata for communication between different tools. The representation should be persistent: no re-generation should be required for every tool. We need a high-level model of the method that supports powerful code transformation. Finally, the representation should be causally connected: changing the representation should change the system with no need of explicitly calling a compiler.

Most of the time, modern OO languages provide two representations for the method level: the text that the programmer typed, and bytecode, the language that the virtual machine interprets. Internally, tools that

need sub-method structure do not use the text nor the bytecode directly, but generate a custom representation, in many cases an *Abstract Syntax Tree* (AST). We now evaluate the three representations as a foundation to support sub-method reflection.

4.2.1 Text as Sub-Method Representation

In all current major programming languages, the programmer types text [46, 54]. This text is then used by the compiler to generate executable code. Text itself has no real structure — it is just a collection of characters. Thus text as a sub-method representation has the following problems:

Low-level. Text does not provide any high-level interfaces: it lacks the possibility to scope information and manipulate the underlying program elements.

Not causally connected. Changing the source code of the method has no effect. We need to call the compiler to generate a method.

Therefore text is almost never used directly for analysis and manipulation. Instead, tools parse the text into an intermediate format such as an AST.

4.2.2 AST as Sub-Method Representation

A commonly used and generated intermediate representation is the AST. For example, ASTs are used by the compiling chain and refactoring engine. Using the AST as sub-method representation has the following problems:

Not persistent. The AST is not persistent: it is generated and then used, but not stored. While trees can be created from source code, the meta-information, which we would like to associate to specific nodes, has to be stored in separate structures.

Not causally connected. Changing an AST does not have an immediate effect on the underlying run-time behavior. Depending on the compiler API, the AST or the method body text is passed to the compiler, which will compile and install the method in its class.

4.2.3 Bytecode as Sub-Method Representation

In contrast to the other representations, bytecode is causally connected: changing bytecode directly changes the behavior of the system. Bytecode

has been used, in both Java and Smalltalk, for a form of structural reflection (Javassist [30] and BYTESURGEON [40])[1]. Both provide a high-level interface to the user, *e.g.*, by abstracting away bytecode details such as the different encodings for message sending or providing a way to specify code to be inlined as a string in the host language. Nevertheless, in the end programmers are forced to deal with bytecode level abstractions which may be different than the programming language the methods are written in [23]. Bytecode as a model for sub-method structure has a number of problems:

Different representation needs. There is a dilemma: on the one hand the execution engine (bytecode VM) requires bytecode for execution and on the other hand programming environments or programmers require text or an abstract and high-level representation of the source code.

Low level abstraction. The programmer has to deal with the idiosyncrasies of the bytecode representation. For example, control structures may be optimized as it is the case in Smalltalk bytecode. In addition, there is a mismatch between the program level abstractions and its runtime [23].

All three discussed representations have the problem of missing extensibility and lack a way to describe metadata:

Missing extensibility. Tools cannot easily extend the representation for their needs. Thus, they often use a custom representation which leads to a situation where tools cannot easily share meta-information, *e.g.*, when one tool gathers information for others to use [130, 103].

Mixed base-level and meta-level code. Code transformation (*e.g.*, bytecode instrumentation [30]) is often used to reflect on method execution: small snippets of code, so called hooks, are inserted into the original code. This leads to the problem of distinguishing code of the original method from the code added by the instrumentation framework.

[1] In the case of Javassist for Java it should be noted that it can only provide load-time structural reflection: newly loaded code can be transformed, but once it is loaded, no further change is possible. In contrast, BYTESURGEON provides full runtime transformation capabilities: methods can be transformed even in a running system.

4.2.4 Requirements

We want the best of both worlds: AST and bytecode representations. In summary, the representation should be (i) causally connected and well integrated in the system, (ii) persistent, (iii) extensible and (iv) reasonably compact with minimal performance impact. Furthermore, it should support the separation of base and annotated code and offer a high-level abstraction to the developers.

Such a model for reflective methods makes it easier to develop and deploy a new generation of tools that work on sub-method elements.

We have seen that bytecode is too low level for our purpose, but the AST looks promising. We need to take practical consideration into account: can such a high-level representation be reasonably compact and efficient? In the next section we present a solution that satisfies these constraints.

4.3 Reflective Methods: Annotated ASTs

Our solution is based on a dual representation of methods which combines an AST-based representation offering high-level manipulations with a compact bytecode-oriented representation supporting fast execution. The abstract syntax trees can be annotated, the semantics of annotation is defined by specializing dedicated compiler plugins. The causal connection and efficient execution is ensured by dual methods that recompile bytecode automatically from their associated and annotated AST. In such a context, we define three meta-objects: (1) the ASTs and their associated transformation API, (2) the annotations and their semantic definition specified by (3) the plugins. This set of objects enables what we call a *reflective method*.

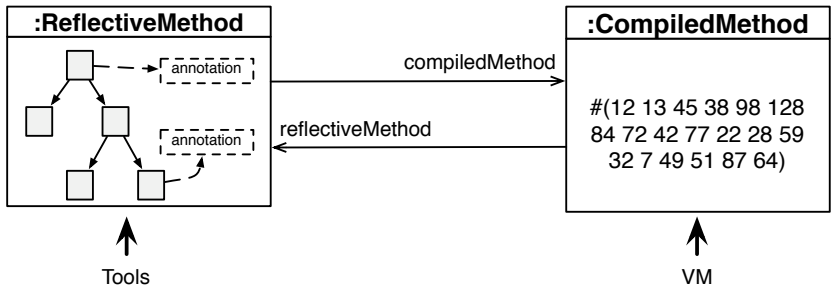

Figure 4.1: A reflective method is the meta-object of a compiled method.

4.3.1 Dual Methods

We implemented the presented approach in Squeak, an open-source Smalltalk. In Smalltalk, code is compiled to bytecode which resides in compiled method objects (an instance of the class CompiledMethod [61]). Besides bytecode, a compiled method keeps a pointer to its source and a dictionary for additional state that is associated with a method (*e.g.*, its name). We enhanced the compiler to generate a reflective method instead of a compiled method. A reflective method provides access to its AST meta-object. Before its first execution, a compiled method is generated from the reflective method. Such a compiled method is cached to minimize performance loss. When the code of a method is changed the cache is flushed and the reflective method is reinstalled in the method dictionary as shown by Figure 4.1 and described in [98].

The system was named after *Persephone*, the greek goddess who spends half of her time in the underworld and the other half in the upper world. Like *Persephone*, methods are seen sometime as part of the underworld (the virtual machine), other time as part of the upper world of high-level abstractions.

Before going into the details of our sub-method protocol we present an example of annotations that are visible in the source code. We show later that some annotations may be invisible in method source code.

4.3.2 A Simple Example: Compile-Time Evaluated Expressions

The following piece of code is the definition in Smalltalk of the method calculateNinePower which evaluates and returns the result of 9 to the power 10000. Without the annotation ($<$:evaluateAtCompiletime :$>$) the execution of such a method will at runtime send the message raisedTo: to the object 9 and then return 9^{10000}. Using annotations we can mark any abstract syntax tree element, *i.e.*, any expression. Here we specify that the expression should be executed at compile-time.

```
calculateNinePower
  ^ (9 raisedTo: 10000) <:evaluateAtCompiletime:>
```

Semantics Definition. At compile-time, the resulting value replaces the annotated expression. The semantics of the annotation is defined by creating a compiler plugin called CompiletimeEvaluator. The complete implementation is based on two classes which specialize the meta-objects and define an annotation and a compiler plugin (see Figure 4.2).

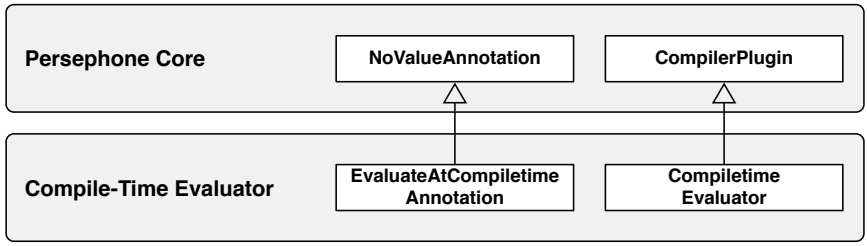

Figure 4.2: Extension for compile-time evaluation.

EvaluateAtCompiletimeAnnotation is a subclass of NoValueAnnotation since this annotation does not expect arguments. The class implements one method: key that returns the annotation name (here the symbol evaluteAtCompiletime).

The compiler plugin class, named CompiletimeEvaluator, is a subclass of CompilerPlugin. Besides that, it only has two small methods visitNode: and evaluateNow:.

```
visitNode: aNode
   ^(aNode hasAnnotation: EvaluateAtCompiletimeAnnotation key)
      ifTrue: [ self evaluateNow: aNode ]
      ifFalse: [ super visitNode: aNode ]

evaluateNow: aNode
      | value literalNode |
      value := aNode evaluate.
      literalNode := LiteralNode value: value.
      aNode replaceWith: literalNode.
      ^self visitNode: literalNode
```

We check every node for the evaluteAtCompiletime annotation. If the annotation is present, we pass the node to evaluateNow: which does the evaluation. If the annotation is not set, we just continue the visiting process. The method evaluateNow: evaluates an expression by sending the evaluate message, a literal node is created to hold the result. We replace the original node and the visiting process continues.

4.3.3 AST and Tree Transformation API

The AST used in PERSEPHONE is an extended version of the Smalltalk refactoring engine AST [114]. We extended it to provide annotations for all node objects. Reflective methods provide a comprehensive abstract syntax tree API. Trees are easily edited and transformed (added/removed/re-

placed, see Table 4.1). On top of these simple transformations, the ParseTreeRewriter is a rewrite engine which allows transformations to be specified at a high-level of abstraction.

Node	Operation	Description
any node	replaceWith:	replace this node
	replaceNode:withNode:	replace a direct child node
sequence node	addNode:	append a node at the end
	addNodeFirst:	insert in front
	addNode:after:	insert after a given node
	addNode:before:	insert before a given node
	removeNode:	remove a node
	replaceNode:withNodes:	replace one node with a collection

Table 4.1: The transformation API for nodes.

It should be noted that this API provides a way to destructively transform a tree: after the transformation, the tree is changed in the same way as if we had edited text and recompiled the method. This is useful *e.g.*, for refactorings. However, destructively changing a tree is not always what we need. As we will present in Section 4.4.1, annotations provide a way to define non destructive transformations *i.e.*, we can always identify the original method.

4.3.4 AST Annotations

As discussed in Section 4.2.4, reflective methods should serve as the main method representation for many different usages. A consequence is that users need to be able to add information about objects (nodes) directly to those objects themselves. Adding behavior to the node classes is possible using the class extension mechanism of Smalltalk which allows a package to define methods on classes defined in another package [10]. For object state extensions, our MOP provides annotation objects that are attached to any node as shown in Figure 4.3.

Each annotation is uniquely identified by a key and each node has a dictionary that maps keys to annotations. Annotations are instances of one of the three subclasses of Annotation. They can have zero, one or multiple values. To define a custom annotation we subclass from the appropriate subclass depending on their number of values (see Figure 4.3). The difference between multi-valued and single valued annotations is that the former can be defined multiple times on the same expression with different values whereas the latter cannot.

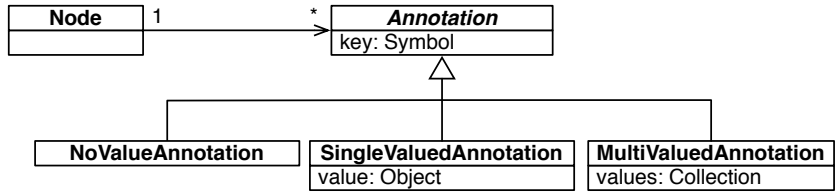

Figure 4.3: The annotation hierarchy.

The expression anExpression <:aSelector: anArgument:> attaches an annotation with one argument to the expression anExpression. Annotations are supported on all expressions and additionally on method arguments, block arguments and each variable name in a temporary variable definition.

The argument of an annotation can be any Smalltalk expression. In the simplest case, when the argument is just a literal object, the value of the annotation is set to this literal object. When the argument is an expression, the value of the annotation is the AST of this expression. We can specify when this AST is evaluated, either at compile time or later at run-time. In addition we provide a reflective interface for annotations which can be queried and set at runtime.

Annotations may or may not appear in the source code. For example, an invisible annotation is the number of times a particular tree element has been executed; an example for a visible annotation is a type declaration. Non-textual annotations can be added reflectively to any node at runtime. These annotations are kept as long as the AST is not regenerated. For example, when a method is recompiled from source, the nodes of this method will have no annotations, but clients can be notified of any code change and add annotations again if needed.

4.3.5 Annotation Semantics

Without specific interpretation, annotations are pure metadata: they have no predefined semantics. To specify annotation semantics, PERSEPHONE defines a meta-object protocol for the compiler and bytecode generator.

The MOP is based on a plugin architecture. Before generating any code, the compiler framework copies the AST and then calls all defined compiler plugins by priority order. A compiler plugin is just a subclass of CompilerPlugin. Plugins affect compilation by transforming the AST. As we provide fully reflective access to the annotations, the compiler plugin may

take annotations into account. We present a full example in Section 4.4.1 with the instrumentation framework TREENURSE.

4.3.6 Characteristics of the Solution

Now we analyze our approach according to the requirements presented in Section 4.2.4.

Causal connection. The implementation ensures that the executed bytecode is always in sync with the reflective method. The whole mechanism is completely transparent to the user. Thus, we provide a causally connected and integrated model.

Persistency. Reflective methods (the AST including the annotations) are installed in the method dictionary of a class. As reflective methods are normal Smalltalk objects, they are written to the disk when the system is stopped. Thus the model is persistent.

Abstraction level. We reuse the same representation and API as the Smalltalk refactoring engine which has proven to be a usable abstraction for various analyses such as refactoring and general metaprogramming.

Separation of base and meta-level code. The annotation framework provides a way to structure code into data (AST) and metadata (annotations). For example, instrumentations as the results of the metaprograms are still completely identifiable since they are represented in the AST as annotations.

Extensibility. Annotations provide a convenient way to associate metadata with the AST nodes for storing additional state. Due to the compiler MOP, we can use annotations to extend the language semantics.

Size and performance We will discuss memory consumption in Section 4.4.3. The provided caching scheme ensures that at runtime we generate a compiled method only once. In situations where even this relatively low overhead is too large, we can statically generate all bytecode before deployment.

4.4 Validation of Sub-Method Reflection

First, we present two case-studies realized with sub-method reflection: an instrumentation framework and a pluggable type system. Second, we discuss memory and performance aspects of the system.

4.4.1 Instrumentation using Annotations

TREENURSE is a framework for instrumenting code. It is designed to have an interface very similar to BYTESURGEON, the bytecode transformation framework presented in Chapter 3. In contrast to BYTESURGEON, it works on the AST and is implemented as a PERSEPHONE compiler-plugin.

To present TREENURSE, we start by discussing a simple example. We have a method that does an assignment and we want to annotate the code with trace statements that increases a counter. The original code and what we want to actually do at runtime are shown in Figure 4.4.

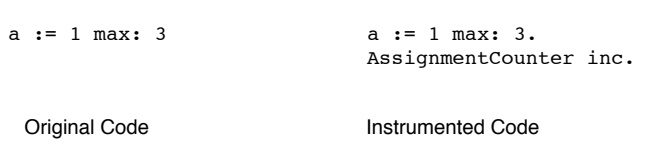

Figure 4.4: Code instrumented with a counter.

It is important to understand that we want to change the semantics of the method, but we do not want to change the code itself. The counter is not part of the design of the system, it is just a temporal addition for debugging. If we transform the code with the help of the refactoring API of the AST, the result would be a new method where the debugging code would be indistinguishable from the original code. Bytecode instrumentation frameworks such as BYTESURGEON or Javassist have the same problem: once the bytecode is transformed, we do not know which statements are part of the original method and which have been added, except at the price of tedious bookkeeping.

An instrumentation framework built on top of PERSEPHONE does better: we use annotations to store the instrumentations in the reflective method and they are taken into account when generating code, as shown in Figure 4.5.

The actual code to do this annotation with our framework looks like this:

```
method instrument: [ :node |
  node isAssignment ifTrue: [ node insertAfter: [ AssignmentCounter inc ] ] ].
```

This code adds an *after* annotation to all assignments in a method with the effect of incrementing the counter. We instrument the method by sending instrument: passing a block with the instrumentation code. We only

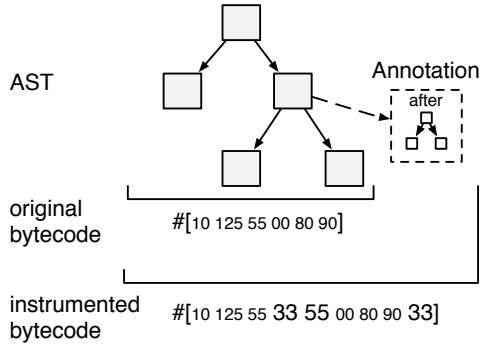

Figure 4.5: Instrumenting a reflective method with annotations.

want to instrument assignments so we select them by sending isAssignment to each node passed to the block.

By implementing TREENURSE using *reflective methods*, TREENURSE has the following unique properties:

- It works on AST nodes and not binary code.
- The instrumentation is stored as an annotation on the AST node. The original AST is left untouched.
- The generation of bytecode is done lazily, on need. Instrumentation merely results in the bytecode cache to be reset. This can lead to considerable time savings especially for large programs.

The Instrumentation API

On all the AST nodes, we support the following transformation API:

Operation	Description
insertBefore:	Annotate to insert code before this node
insertAfter:	Annotate to insert code after this node
replace:	Annotate to replace the node by the code

For an easy selection of the nodes to be instrumented, we provide dedicated iteration interface, for example instrumentAssignments: iterates over all assignment nodes.

Meta-Variables

The only variables that can be directly referenced from inside the block are self, super and thisContext, as the code of the block is to be inlined into the instrumented method. Thus, these variables will be bound to their values at runtime, not at instrumentation time. For everything else, block arguments have to be used. Each kind of node provides a different set of meta-variables that can be used as block arguments:

Node	Meta-variable	Description
any node	node	the node itself
message node	receiver	the receiver of the message
	arguments	a collection of all arguments
	selector	the selector of the message
method node	arguments	a collection of all arguments
	selector	the selector of the method
assignment node	variable	the variable to be assigned to
	variableName	the name of the variable
	value	the expression to be assigned
return node	value	the expression to be returned
variable node	value	the value of the variable
literal node	value	the value of the literal

The following code gives an example of a replace annotation, which uses two meta-variables, one referencing the variable and the other the value of the assignment:

```
method instrumentAssignments: [ :node |
   node replace: [ :variable :value | variable := value + 3000 ]
```

TREENURSE replaces all assignments with a new assignment that adds the number 3000 to the value expression of the original code.

Example: Code Coverage Analysis

Code coverage analysis per expression is conceptually a simple task. When an expression gets executed it is marked as executed. After the program is run the executed expressions are printed differently from the ones that were not executed. The most convenient way to store information about a node is by adding an annotation to it that holds the information. To keep track of how many times a node has been executed we create a subclass of SingleValuedAnnotation named ExecutedAnnotation. We then add a method markExecuted to ProgramNode via a class extension:

```
Node>>markExecuted
  (self annotationAt: ExecutedAnnotation key) increment
```

If we now send markExecuted to any node, its execution count will be incremented, which we can achieve using our instrumentation framework:

```
method instrument: [ :eachNode |
    eachNode insertBefore: [ :node | node markExecuted ] ].
```

Here we iterate over all the nodes of the method, the block is evaluated for each node, binding the node to the variable eachNode. Before each node, we insert code. The inserted code is described by a block. This block references the meta-variable node which references the AST node at runtime.

Here we see a usage of the node meta-variable. It reifies the node, so we can call the markExecuted directly on the AST nodes. To produce the final output a pretty printer presents the status of execution.

Usage of TREENURSE

TREENURSE has seen some use by other projects, especially for dynamic analysis. It has been used for test coverage analysis [109] as an alternative to BYTESURGEON. Adrian Lienhard used it in an experiment that analyzes how objects flow through an object-oriented program at runtime [95, 93, 94].

4.4.2 Pluggable Type System

TYPEPLUG [67] is an optional, pluggable type system [22] for Squeak. It consists of a type reconstructor and inferencer that is used by a type checker to check Squeak programs for type correctness.

A type checker is an example of a program that works on code: we need to be able to attach metadata to expressions (the types) and be able to reason about this metadata. Thus the realization of a type system validates especially the extensibility of the reflective methods. We need to be able to model metadata that describes the type of any expression in the code.

With PERSEPHONE, types are represented as annotations on nodes in the AST. They can be declared on method and block arguments, method and block return values and temporary variable declarations. There are two different ways to declare types.

1. Using a special code browser to annotate the nodes. This has the advantage that types are declared without changing the source code

but can still be checked into a source code management system. This is the preferred way for typing existing code especially system classes like Boolean or Collection.

2. Textual annotations, which are placed in the source code.

The following code shows a method annotated with types:

```
bitFromBoolean: aBoolean <:type: Boolean :>
        ^ (aBoolean ifTrue: [1] ifFalse: [0]) <:type: Integer :>
```

The method takes a boolean as an argument and returns an integer.

The TYPEPLUG case study shows that our representation provides the extensibility needed for extending the language with a pluggable type system. TYPEPLUG demonstrates the usefulness of providing textual annotations where the annotation itself is not limited to be a static predefined datatype. PERSEPHONE supports annotations that contain general Smalltalk code. The evaluation of this code can be completely controlled by the annotation class itself or the compiler plugin. This allows for building complex annotations as required *e.g.*, for advanced type systems.

This is only a very short introduction into TYPEPLUG. The master's thesis of Haldimann [69] and a conference paper [67] provide more in-depth information. A journal article is in preparation [68].

4.4.3 Performance and Memory Analysis

In this section we discuss results of performance benchmarks and report on memory consumption. The machine used is an Apple MacBook Pro (2Ghz Intel Core Duo, 2GB RAM).

Performance of Cache

The dual method approach works like a cache: after the first run, the compiled method is installed in the class and thus the method can be executed at full speed.

To analyze cache performance, we use the *TinyBenchmark* suite that is part of the normal Squeak distribution (Table 4.2). The TinyBenchmark suite tests bytecode interpretation and message send performance. For this test, we use the runtime of the benchmarks for assessing cache performance. First, we record the runtime for an unmodified Squeak. Then we run the benchmark with PERSEPHONE in two cases: with and without caching the generated bytecode.

When running the benchmark with caching disabled, the system gets too slow to be usable as bytecode needs to be generated for each method execution. We had to abort the benchmark run after one hour. We see a noticeable speedup as soon as we turn on caching: PERSEPHONE shows no detectable slowdown compared to standard Squeak, even though bytecode had to be generated for the benchmark methods on the first execution.

Caching Scheme	runtime
unmodified Squeak	6.9 seconds
Persephone, no cache	>1 hour
Persephone, cache	6.9 seconds

Table 4.2: The effect of method caching.

Thus we can see that the cache provides a substantial speedup and enables a system using reflective method to be as fast as the standard Squeak system.

Memory Considerations

Representing a method as an AST in which each node is an object obviously requires a lot more memory than its corresponding compiled method which only stores the bytecodes. Table 4.3 shows the memory consumption of the AST of a complete Squeak system.

Name	number of classes	memory
Squeak 3.9	2244	20 MB
Squeak 3.9 AST	2244	123 MB

Table 4.3: Memory consumption.

We see that the system uses a lot of memory, but in a typical development scenario, the system is already usable as is without any further optimization. In addition to that, reflective methods are mostly only reified for parts of the system, for example a single package that needs to be analyzed.

To assess if the size is practically usable, we compare to the size of code loaded into Eclipse, a development environment used widely in industry. We took the source of ArgoUML[2] and loaded it in Eclipse version 3.2. ArgoUML Version 0.24 consists of ca. 1300 classes, it is thus much smaller than

[2]http://argouml.tigris.org/

Squeak. Eclipse allocates after startup ca. 90MB of memory and ca. 180MB when having the ArgoUML source loaded. Thus the memory consumption of PERSEPHONE is typical for a modern development environment.

Also note that those figures have to be considered as upper bounds since we have not yet fully applied some memory related optimizations. For example, the size of the AST data can be optimized further by not referencing scanner-token data. In addition to that, we plan to experiment with AST specific compression techniques [58].

4.5 Other Systems

One interesting question is how to realize *sub-method reflection* for other languages than Smalltalk. We discuss two implementation strategies.

Dual methods

We used *objects-as-methods* [6] and the fact that we can replace methods at runtime. Other bytecode based systems could use the same scheme if the virtual machine would be extended to support both features. The *objects-as-methods* feature can be replaced by so called *stub-methods*: we could generate short methods that contain the code that is now implemented in the run:with:in method of the ReflectiveMethod class. The use of stub-methods would be a way to realize sub-method reflection for example in other Smalltalk systems, for example Visualworks without the need to modify the virtual machine.

For other systems (*e.g.*, Java), what is needed is the possibility to replace code of methods at runtime: the system needs to support real structural reflection down to the level of methods. As soon as this is the case, it is possible to realize sub-method reflection very similar to the realization described in this chapter. The ability to change methods at runtime limits the languages in practice to those that either are completely interpreted or provide some form of virtual machine where we can change the code executed at runtime.

Just-In-Time Compiler

Modern virtual machines usually are based on dynamic compilation. Before execution, the bytecode is compiled to binary code that is executed directly by the hardware. The dual-methods implementation strategy works for such a virtual machine without any change: the AST is compiled to bytecode and cached, this bytecode then is compiled to binary code by the virtual machine. The bytecode in such a system would serve only as the interface between the AST and the JIT compiler of the virtual machine.

Sub-method Reflection replaces bytecode as the representation for sub-method structure as far as reflection is concerned. We keep bytecode for the sole purpose of execution: it is the representation the virtual machine works with. For Squeak, which uses a very basic bytecode interpreter, this makes sense. But for systems with *just-in-time* compiler based virtual machines the bytecode is transformed into binary code before execution.

For such a virtual machine, bytecode is not the representation of execution, we execute only the code that the JIT-Compiler generates. Thus for a system with a compiler-based virtual machine, there is actually no need for bytecode: we could directly use the sub-method structure as an input for the runtime compiler that generates binary code for execution. The binary code would directly replace the cached bytecode of our dual methods implementation strategy.

Our solution for Squeak was driven by the objective to actually not change the virtual machine and by the fact that Squeak is a pure bytecode interpreter. There has been work done in the past on a just-in-time compiler for Squeak [38]. An interesting experiment would be to build a system where the high-level sub-method representation is used as the input for the runtime compiler. Such a runtime-compiler based implementation of sub-method reflection would be an interesting implementation strategy for many systems besides Smalltalk, but it requires extensive changes to the virtual machine.

4.6 Related Work

Annotations

Annotations are not new. For example, Javadoc tags are a form of annotation. More recently, Java 1.5 adds support for annotations to the language. As of release 5.0, Java has a general purpose annotation (also known as metadata) facility that permits you to define and use your own annotation types. The facility consists of a syntax for declaring annotation types, a syntax for annotating declarations, APIs for reading annotations, a class file representation for annotations, and an annotation processing tool. Java 1.5 annotations are only allowed for type, field, variable and method declarations and are not allowed on type parameters or method invocations. VisualWorks and Squeak support method annotations (also called pragmas).

Spoon [106] is an open compiler for Java that provides compile-time reflection. With Spoon, the Java AST can be transformed before it is compiled to bytecode. The *processors* of Spoon are similar to the compiler-plugins

of PERSEPHONE. Spoon provides support for Java annotations, *i.e.*, the transformation processors can read annotations. The main difference to our work is that Spoon works at compile-time, not runtime. The AST is compiled to bytecode and is not available at runtime. Spoon provides an annotation-aware compile-time transformation framework, but not causally connected structural reflection at runtime. In addition to that, it is restricted to the annotation model provided by Java.

Higher Level Abstraction: Beyond Text

There have been a number of proposals over the years to move away from text as the only representation of code. Dimitriev [46] argues that programs should no longer be text but a graph described with a meta-model built for a certain kind of problem. The language would be mapped to another one for execution or interpretation. Edwards [54] argues that programs should no longer be text and the representation of a program should be the same as its execution. His programs are trees created by copying. He also identifies the need to customize the presentation of a program. Black [14] makes a case to free programs from their linear structure and replace them with a much richer abstract program structure (APS) that captures all of the semantics, but is independent of any syntax. Conventional one and two dimensional syntax, abstract syntax trees, class diagrams, and other common representations of a program are all different "views" on this rich abstraction. None of these proposals discuss the use of the high-level representation for reflection.

Sub-method Structure

In LISP [101] source code is itself made up of lists. As a result macros can manipulate it using the list-processing functions available in the language. This functionality is limited to macros at compile time and cannot be applied to functions at runtime.

In IO [37] code is a runtime inspectable and modifiable tree. Message arguments are passed as expressions and evaluated by the receiver. Selective evaluation of arguments can be used to implement control flow. IO does not yet provide a way to extend the representation.

KSL [81] represents all language constructs as objects. The representation described includes sub-method elements as objects, it is thus similar to our model. But KSL is purely interpreted, the representation is not compiled. Another difference is that the representation is not extensible and not used to provide behavioral reflection.

4.7 Summary

In this chapter we analyzed the problems of finding a suitable model for sub-method structural reflection. With an extended AST we found a representation that fulfills all noted requirements. We described how to realize this model in a bytecode based environment and discussed memory use and performance characteristics of the approach. We validated sub-method reflection by providing examples of how it is beneficial for tool building (code coverage) and language experiments (type system). In the next chapter we revisit *partial behavioral reflection* in the context of *sub-method reflection*.

Chapter 5

Behavioral Reflection Revisited

5.1 Introduction

In this chapter, we present how we realize *partial behavioral reflection* on top of sub-method structural reflection.

Initially we revisit the problems that exist with our first bytecode-based realization (GEPPETTO 1) presented in Chapter 3. We introduce our new model and then discuss in detail the key aspects that differentiate our new approach from the bytecode-based solution. We provide benchmarks to compare the new implementation (GEPPETTO 2) to the old bytecode-based solution and conclude with a discussion how our new model solves the problems identified in Chapter 3.

5.2 The Problems

We discussed in Chapter 2 and have shown in detail with our first version of partial behavioral reflection (Chapter 3) that there are some problems with realizing partial behavioral reflection with bytecode manipulation.

We identified three main problems:

Semantic mismatch. Concepts existing at the level of the language are not represented at the level of the bytecode. In the case of Smalltalk, examples are blocks and message sends encoding control structures.

Code quality. The stack based nature of bytecode make it hard to generate optimized code.

Problem with synthesized elements. To realize behavioral reflection, we modify the bytecode. It is problematic to distinguish between this synthesized code and the base level code.

Initially we present our new approach and then show how it solves the above problems.

5.3 Partial Behavioral Reflection Revisited

The central notion of partial behavioral reflection is the *link*. The link defines the meta-object of operations and defines the protocol between the base-level and the meta-object. In the classical Reflex model, operations are not directly bound to the meta-object. Instead, operations are combined to *hooksets*. Hooksets are sets of bytecode-level operation occurrences. They denote the place in the bytecode where code, the so-called *hook*, is added that calls the meta-object. The hookset is bound to the meta object using the link, as shown in Figure 5.1.

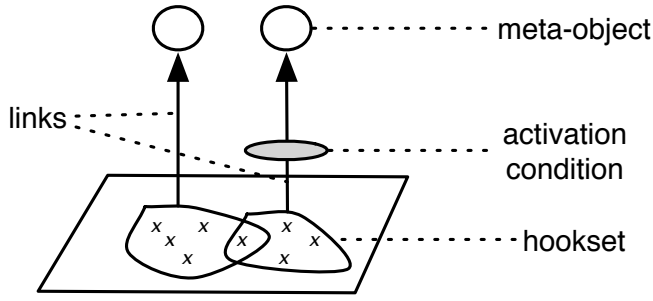

Figure 5.1: Links, hooksets and meta-objects in the original model.

With sub-method reflection, we have a structural representation of method bodies. Operations have a direct representation as part of the sub-method structure: they are nodes in the AST. In addition, the AST represents the structure of the method, for example both the block structure and the structure of expressions are represented. We can adapt the model of GEPPETTO to use sub-method reflection for selecting the operations to reify instead of using hooksets. In addition to operations, as the AST

represents the complete structure of the method, we can for example reify block evaluation.

To summarize:

- We use sub-method reflection to select what to reflect on.
- Instead of defining a hookset, we annotate the AST directly with a link.
- The hookset is implicitly defined to be all nodes with the same link annotation.

Figure 5.2 shows how the AST, links and meta-objects work together in the new system.

Links are annotations on the representation provided by sub-method reflection. When we set a link, our system automatically regenerates the bytecode the next time a method is invoked. The regeneration of bytecode will take the link into account, as there is a compiler plugin that transforms the code as specified in the link. We provide a more in-depth discussion of the implementation in Section 5.5.

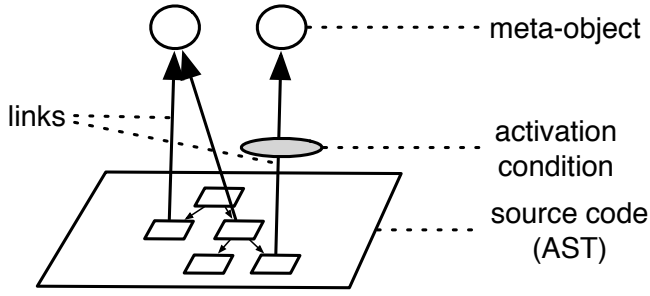

Figure 5.2: Partial behavioral reflection realized with sub-method reflection.

Dynamic Properties

By realizing GEPPETTO on top of *sub-method reflection*, we can exploit the dynamic properties of the PERSEPHONE model. Any change on the structure, in this case setting a link, will result in new code being generated on the next run of the method. The result is that we can install or uninstall existing or newly created links at any time, even while the system is running.

In addition, the link uses the same mechanism to report changes of itself. All links are part of the *causally connected* representation.Therefore, we can change existing links dynamically. If we change any of the attributes of the link, for example the meta-object, all methods where this link is installed in will be recompiled prior to the next execution.

As with GEPPETTO 1, changes are activated on a per-method basis. After a link is set, the next invocation of the method will lead to new code being generated. While code is generated, the old method is active, it is atomically exchanged with the new method. So in principle, we have the same per-method granularity of introducing links as seen in GEPPETTO 1.

5.3.1 Simplifications

This section describes the simplifications of our new approach as compared to the bytecode-based approach described in Chapter 3. Most of the simplifications are related to introducing classes only when really needed. Many classes existing in the first version have either been merged with the class Link or have been replaced by simple literal objects like symbols.

No hookset. The first simplification comes directly from the fact that we do not have the concept of a hookset: links are inserted directly as annotations on the nodes of the AST.

Symbols instead of classes. Link attributes are symbols whenever possible. This reduces the number of classes (*e.g.*, the Parameter class) and simplifies the code that needs to be written when defining a link.

Condition is a block. The condition attribute can be any object that supports the protocol of the block-closure object for evaluation. In this way, we can in most cases use a block as the condition; for more complex cases special classes can be provided.

No *id*. Links do not have a human readable id.

No CallDescriptor. The CallDescriptor class has been removed as the description of the protocol between the base the meta is contained in the link itself. We only support what used to be the old PassingMode plain where the reifications requested are passed directly as arguments to the meta-object. Passing via an array (PassingMode array) just adds unnecessary overhead due to array creation.

Figure 5.3 shows the resulting simplified model.

In the following section, we will describe the link and its parameters, then we briefly discuss access to reified data.

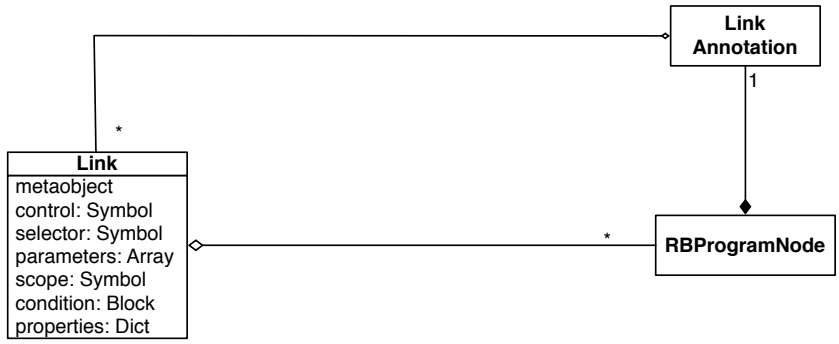

Figure 5.3: The link in GEPPETTO 2

5.3.2 The Link

In our new model, the *link* is an even more central abstraction than in the original model. The link defines where reification occurs by being set as an annotation on the AST nodes, it defines which meta-object to call and it contains the definition for the exact protocol between the base and the meta-level.

The parameters of the link are contained in attributes that are set when configuring the link prior to installing it in a node. The attributes are set via single message sends to the link object. For convenience, default values are defined for the attributes.

To illustrate this, we show as an example a link that, when installed on a message send node, would replace this send with an invocation of a method #send:to:with: in the meta-object of class Sender:

```
link := GPLink new metaObject: Sender new;
      selector: #send:to:with:;
      arguments: #(selector receiver arguments);
      control: #instead.
```

Link Attributes

In the following, we provide an overview of all the attributes that can be configured for a link.

Meta-object. Primarily, the link defines which meta-object to use. This can

be *any* Squeak object. There is no special superclass or any special structure or methods required.

```
link metaObject: Sender new.
```

Selector: The selector defines which method to call on the meta-object. Thus, it is a symbol; the default is #value. When the selector defined requires multiple arguments, these are defined with the #arguments: attribute.

```
link selector: #send:to:with:.
```

Arguments. When calling a method with multiple arguments, we have to specify which arguments to pass to the meta object. The arguments are specified by a literal array containing symbols describing what exactly to pass as arguments. The default is the empty array.

```
link arguments: #(selector receiver arguments).
```

We use symbols to specify which information exactly is to be passed to the meta-level. We give a complete list later in the section on *reified data*.

Control: The *control* attribute specifies when the call to the meta-object should occur: *i.e.*, before, after or instead of the original operation. The control is specified using a symbol. Possible values are #before, #after or #instead. The default is #before.

```
link control: #instead.
```

Condition. Optionally we may want to control the link further by specifying a condition. This condition can be any object that returns a boolean value when being sent #value. This means we can use Booleans, as *e.g.*, true value returns true. Blocks are the most natural condition. In addition, the user of the framework can provide a special object that implements #value if needed.

With blocks as conditions, link activation can be easily controlled. As an example, the following link would only be active when tracing would be turned on in the system-wide preferences:

```
link condition: [Preferences traceEnabled].
```

The link is active if the block evaluates to true, that is, when the preference is enabled.

For conditions that depend on reified information only available at runtime from the base object, we provide a way to specify which parameters to pass to the block by using #condition:arguments:. We can pass any reified information to the condition block. The following would restrict a link to one certain object:

```
link condition: [:object | object == myObject] arguments: #(object).
```

Meta-object Scope. As in GEPPETTO 1, we provide the concept of meta-object scope. This enables us to have different meta objects associated for different base level entities instead of one global meta-object per link. We can configure a link to have meta objects *per class*, *per method*, *per object* or *per node*.

```
link metaScope: #object.
```

When using meta-object scope, we have to create new meta-objects at runtime. Therefore, we specify the meta-object for the link not statically, but using a block of code:

```
link metaCreator: [MyMeta new].
```

The system takes care to create new meta-objects as required using this block and it manages the association of meta-object to the entities requested.

Concurrency. Optionally, we can parametrize a link to execute a meta-object in its own thread.

```
link beConcurrent.
```

The link calling code is wrapped in a block that is then sent the message #fork to start a new process. We have not fully explored the potential to incorporate concurrency aspects in our MOP. We identify a potential to address these aspects in future work.

Inlining Properties. The last set of attributes that we describe are those related to code generation. They have no semantic meaning; they only change performance properties.

Both the meta-object and the condition can be either inlined in the bytecode or accessed via an indirection over the link. Either of those options has performance properties that can be beneficial depending on runtime usage. The default behavior is to inline both condition and meta-object.

When we change the meta-object or the condition of existing, installed links at runtime, all affected methods need to be recompiled. Therefore, if we plan to change the meta-object at runtime frequently, we can configure the link to be non-inlining. The unfortunate side-effect of that is a slight decrease in link-call performance, as any call to the meta-object (or the evaluation of the condition) is indirected via the link.

```
link inlineMeta: false.
link inlineCondition: false.
```

The decision not to inline should only be taken when it is sure that changes happen so frequently that recompiling would have a larger effect on performance than the indirection. An interesting property of our fully dynamic model is that we can switch the inlining behavior at runtime. There is no need to decide this up-front or have a link fixed to one behavior over its entire lifetime.

Access to Reified Data

We support various kinds of reified base level information that can be passed to the meta-level. Examples are the arguments passed to the method that is called on the meta-object. Table 5.1 shows all reifications available in the standard system. All these are defined using a plugin based architecture and can be thus extended by any user of the framework (see Section 5.5.2).

Many of the reifications are already mentioned in Chapter 3 such as the receiver and arguments of a message send. We have added some that open up interesting possibilities:

Link. We can request the link to be passed to the meta-object. In this way we can for example deactivate links from the meta-level or reflect in general on links.

Node. The node that the link is installed on. This allows the meta-object to reason about or even to annotate or modify the structure of a method. Later in this chapter we see an example how this provides an easy way to implement a tracing tool for dynamic analysis.

Continuation. Sometimes, we want to abort the execution of the meta-level and return to the base. We can request a *continuation* to be created and passed to the meta-object. The continuation is an object that, when sent `#value` will continue execution of the base level code. The continuation is a true continuation. We based our implementation on the realization of continuations in Seaside, a continuation-based

Operation	Reified Data	Description
All Nodes	#context	current stack-frame
	#object	the object
	#class	the class of the object
	#control	before, after or instead
	#link	the link itself
	#node	the node that the link is installed on
	#continuation	a continuation object
	#process	the current thread executing
Message Send/ Method Evaluation	#arguments	arguments as an array
	#argX	X^{th} argument
	#sender	sender object
	#senderSelector	sender selector
	#receiver	receiver object
	#selector	selector of method
	#operation	operation as object
	#result	returned result (after only)
Block	#arguments	arguments as an array
	#argX	X^{th} argument
	#result	returned result (after only)
Temp/InstVar Read	#varname	name of variable
	#offset	offset of variable
	#value	value of variable
	#operation	operation as object
Assignment	#varname	name of variable
	#offset	offset of variable
	#value	value of variable
	#operation	operation as object
	#newvalue	new value (write only)

Table 5.1: Supported reified data.

web-framework [51, 12]. We can, for example, store the continuation object in the meta-object before we return normally to the base level. Later we can use the stored continuation to return from the meta-level again as many times as we like.

We specify the information to be passed to the meta-level with literal symbols which results in far more concise code then the dedicated Parameter class used in Chapter 3.

Links are not created to be specific to a certain kind of node. This is important to provide the possibility of crosscutting not only class boundaries but even operations: we want one link to be installable on both message-sends and assignments, for example. But not all nodes support the same

reified data. For example, only message sends have a receiver. We therefore check at the time that the link is set as an annotation on a node if the link is compatible with the node.

5.3.3 Spatial Selection

One of the major advantages of the AST as an underlying model for behavioral reflection over the bytecode-based approach is that the AST faithfully represents every concept of the programming language, whereas at the level of the bytecode some optimizations already have been applied.

The AST has a clear advantage in the following cases:

Variables are represented as nodes in the AST. In addition, assignments have their own representation. We can put links on variables and assignments. All types of variables are represented, including global and class variables.

Blocks written in the source code are represented at the AST level, we can annotate them with a link. When a link on a block is configured to have the control #after, the system will use exception handling to make sure that the *after*-code is even evaluated when the block is aborted due to abnormal termination or early returns.

Control Structures like ifTrue:, whileTrue: or to:do: are not message sends on the bytecode level anymore, they are optimized to be encoded as jumps for performance reasons. On the level of the AST, all these are standard message sends. For our system, this means that we can put links on the message sends of control structure and on the blocks that are the parameters of these control structures.

Selecting Nodes

The structural model provided by sub-method reflection has static representations for everything that a behavioral reflection framework needs: all operations (for example, message sends), the methods themselves and even block objects. Therefore, we can directly use this representation for *Spatial selection*.

The nodes of the sub-method AST are directly annotated with a *link*. This means that we use the standard structural reflection of Smalltalk to iterate over classes and methods. Then we iterate over the sub-method structure, the AST. Setting the link is done by sending the message link: to a node.

Message	Description
nodes	all nodes
sends	all message send nodes
blocks	all block nodes
statements	all nodes representing statements
assignments	all variable assignments
variables	all variable nodes
variableReads	nodes of variable reads
variableWrites	nodes of variable writes
instanceAssignments	instance variables
instanceVariableReads	
instanceVariableWrites	
tempAssignments	temporary variables
tempVariableReads	
tempVariableWrites	

Table 5.2: Convenience methods for spatial selection.

To make iteration easier, we provide convenience methods on both methods and classes that iterate the AST and return a collection of nodes. For example, to annotate all message sends in the method Object»#halt we send the message sends which returns a collection of all send nodes of the method:

```
(Object>>#halt) sends do: [:send | send link: myLink].
```

Table 5.2 shows all convenience methods supported by both classes and methods.

Discussion

Compared to the specification of hooksets as described in Chapter 3, our new scheme has both positive and negative aspects. On the positive side, we can set one link across multiple kinds of operations. The fact that the meta-object can span multiple classes and even different kinds of operations is one of the interesting features of partial behavioral reflection. This allows, for example, to define a counter as a meta-object and define one link that then is installed on multiple classes, recording both variable accesses and message sends. In GEPPETTO 1, defining hooksets that cover different operations used composition of hooksets that where specific to one kind of operation. With GEPPETTO 2, there is no need to build hooksets by composition. We can set a link as an annotation on any node in the system,

as long as the link only requests those parameters to be passed to the meta that all the nodes can provide.

On the negative side, spatial selection is procedural: we write code that iterates over the structure and sets annotations on the selected nodes. What used to be the *hookset* is implicitly defined to be those nodes where a link is set as an annotation. The problem is that we lose the ability to easily support the case when the system is changing. Links are not automatically applied to changed or newly loaded code.

This is a shortcoming, but in practice, support for re-installing links in new code can be implemented explicitly by the client using the framework. We plan to extend spatial selection with a declarative way to specify where links are supposed to be installed, bringing back the declarative nature of the hookset definitions.

5.3.4 Special Meta-objects

We now discuss some special objects that can be used as meta-objects. We discuss the notion of treating blocks as meta-objects, thus providing a powerful code-inlining framework built on partial behavioral reflection. The other topic is the use of reified data as meta-objects.

Blocks as Meta-objects

Meta-objects do not need to be instances of special classes: any object is possible. One very interesting kind of object to use as a meta-object is the *Block* object provided by the Smalltalk language.

Blocks define a piece of code that is evaluated later by sending the message value. Here is an example of a link with a block as a meta-object:

```
link := GPLink new metaObject: [Beeper beep];
      selector: #value.
```

The code of the block can be executed before the node where it is installed. Blocks as meta-objects thus provide the functionality of a code-instrumentation framework similar to the TREENURSE system as described in Chapter 4 and to BYTESURGEON (Chapter 3).

We can easily pass runtime information to the block (what used to be the meta-variables in TREENURSE or BYTESURGEON):

```
link := GPLink new
        metaObject: [:object | object color = Color red ifTrue: [Beeper beep]];
        selector: #value:;
```

```
    arguments: #(object)
```

The code generated for such a link is efficient (see Section 5.5). A TREENURSE like system supporting block-semantics for to-be-inlined code generates exactly the same bytecode. In addition, as this is a standard link, we can control the link using a condition, allowing for easy control. Therefore we have the full power of a code-inlining model, but in addition full control via links and conditions.

What was once the basis for realizing behavioral reflection can in our new system be realized trivially. Block meta-objects subsume the functionality of an instrumentation framework like BYTESURGEON and TREENURSE.

Using Reified Data as Meta-objects

An interesting extension to the original model, we allow reified data to be directly requested as a meta-object. To define such a meta-object, we put a symbol describing the reified data as the meta-object:

```
link := GPLink new metaObject: #class.
```

The meta-object is the reified value #class, which stands for the class of the object at runtime in whose method the link is installed on a node. The same link thus can have different meta-objects, depending on the parameter object requested. It can be different per link installation, or even, as with the class parameter, different for the same installed node for different executions.

We will now give two examples how this can be useful in practice.

Meta-class. MetaclassTalk [17] uses the CLOS [88] inspired model where the MOP is realized by having all operations (message sends, instance variable access) be realized by a method that is implemented in the meta-class. This allows the behavior to be changed on a per-operation, per class granularity. This meta-class based MOP fits well into the link-meta way of looking at behavioral reflection: the meta-object is the class of the method where we install a link on one of the instructions. We pass all the information needed to the meta-object, for example for a message send, we need the selector, the arguments and the receiver.

As an example, for message sends a link would look like this:

```
link := GPLink new metaObject: #class;
            selector: #send:to:with:;
            arguments: #(selector receiver arguments)
            control: #instead.
```

If we install this link on all message sends, the system will always put the class of the object where the link is installed in as the meta-object. The method send:to:with: would have a default implementation that implements the normal message send. By overriding this method in a meta-class, we can change the semantics of message sending on a per class basis.

At first glance, one could think that the feature of allowing reified data as meta-objects is not needed. We could just statically generate a link for each class we install the link in. This link then would reference the class statically, a method in class Object would have Object as its meta-object. This is problematic as soon as we consider inheritance. When executing a method of Object as part of a subclass Beeper, the class of this object is of course not Object, but Beeper. Therefore, when we want to have the class of an object be the meta-object, the link needs to dynamically assign the class of the current object at runtime, not the class where the method is defined. The meta-object is the result of the expression self class.

Together with additional links for instance variable read and write, this results in a fairly complete and efficient implementation of meta-class based MOP in the spirit of MetaClassTalk. See Section 5.4.3 for a full implementation.

Node. An interesting meta-object is the node that the link is installed on. Similar to meta-classes, the node is a structural meta-object. For an operation, the node forms the natural static meta-object. For example, it facilitates the implementation of tracing or code-coverage tools by making it possible to invoke a method on the node itself that tags the node that is has been executed:

```
link := GPLink new metaObject: #node;
            selector: #markExecuted.
```

We show an example in Section 5.4.1.

5.4 GEPPETTO 2: Examples

We discuss three examples: implementation of code coverage, realizing method wrappers and a full meta-class based MOP.

5.4.1 Code Coverage

In Section 4.4.1 we discussed how sub-method reflection is useful for tool-building. We illustrated this with an example implementation of a simple code-coverage tool using TREENURSE. Now we show how we can achieve the same thing with partial behavioral reflection. It is possible as we can reify the node that a link is installed on as the meta-object as shown in Figure 5.4.

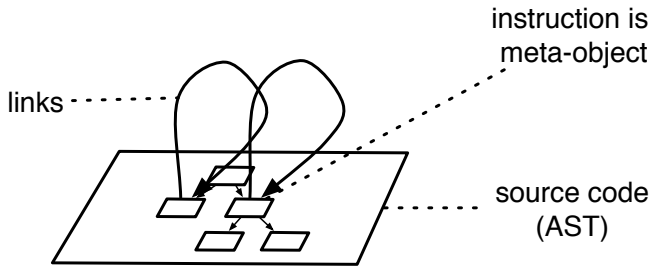

Figure 5.4: Nodes as Meta-objects

For code-coverage, we define a link that calls markExecuted on the node where it is installed:

```
link := GPLink new metaObject: #node;
        selector: #markExecuted.
```

The method markExecuted annotates the node:

```
Node>>markExecuted
  (self annotationAt: ExecutedAnnotation key) increment
```

When we install the link on the node representing methods, we can see coverage at a method level: all methods tagged have been executed. But with our sub-method model, we can go a level deeper and install the link for example on all blocks or even on all nodes of the tree.

To speed up execution, it is even possible to remove the tagging-link at runtime just after tagging the node. In this way, the method would, at the next execution, be recompiled to only call markExecuted on those nodes that have not yet been executed before. In addition, we can leverage the advanced control that the link provides to only activate in special cases, for example to only tag a node when executed, for example, when code is executed due to the unit-test framework.

5.4.2 Method Wrappers

Method wrappers [24] are a common technique to implement behavioral reflection. They have been used for dynamic analysis, to realize dynamic *Aspect Oriented Programming* (AOP) [77] and *Context Oriented Programming* (COP) [78]. We used method wrappers already as an application to validate our BYTESURGEON framework in Section 3.2.5.

Method wrappers define *before* and *after* code to be called around the original method execution. For this, the original method is replaced by a stub-method sending the message #valueWithReceiver:arguments: to the wrapper, which in turn calls the before and after method:

```
MethodWrapper>>valueWithReceiver: anObject arguments: args
    self beforeMethod.
    ^ [clientMethod valueWithReceiver: anObject arguments: args]
        ensure: [self afterMethod]
```

Users of method wrappers implement a subclass of the MethodWrapper class and provide their own before and after methods.

Instead of generating a method that forwards to #valueWithReceiver:arguments:, when realizing method wrappers with GEPPETTO, we can define links that call the before and after methods and install these links on the original method:

```
GPMethodWrapper>>install
   (self class includesSelector:: #beforeMethod) ifTrue: [
      beforeLink := GPLink new metaObject: self;
                                selector: #beforeMethod;
                                control: #before.
      self methodNode link: beforeLink].
   (self class includesSelector:: #afterMethod) ifTrue: [
      afterLink := GPLink new metaObject: self;
                                selector: #afterMethod;
                                control: #after.
      self methodNode link: afterLink].
```

We only install a link when there is a before/after method defined in the wrapper. This is especially important for the after method, as we wrap the complete method in an exception handler to make sure that the after part is executed in any case.

The resulting code is more efficient than the original MethodWrapper implementation, as the call to the before and after methods are inlined in the wrapped method. The code is only slightly less efficient then the solution based on BYTESURGEON presented in Section 3.2.5. With BYTESURGEON, we inline the code of the before/after methods into the wrapped

Method Wrapper implementation	Installation time (ms)	Installation factor	Runtime time (ms)	Runtime factor
Hancoded	-	-	10161	1
Standard	1102	1.0	28443	2.8
BYTESURGEON	13835	12.55	10305	1.01
GEPPETTO 2	3354	3.04	10917	1.07

Figure 5.5: Comparing installation and runtime performance of method wrapper implementations.

methods, whereas with GEPPETTO we inline calls to these methods.

Benchmarks for MethodWrappers

To show the performance of our new solution, we repeat the benchmark as presented in Section 3.2.5. Figure 5.5 shows the result. Our new implementation is faster by a factor of four compared to the BYTESURGEON implementation while the execution speed remains practically the same.

5.4.3 Meta-class MOP

In an MOP like MetaClassTalk [17], the meta-class is the behavioral meta-object. It defines the semantics of message sends, for example.

To realize a meta-class based behavioral MOP with GEPPETTO, we need to reify message sends and let the class object do the message send instead of letting it be executed by the virtual machine. As classes are objects, they have a meta-class that defines this behavior for the class.

We thus define a link where the meta-object is the class of the object the link is installed in. The link calls the method send:to:with: and provides it with all information needed to reflectively do the send:

```
link := GPLink new metaObject: #class;
      selector: #send:to:with:
      arguments: #(selector object arguments)
```

The class Behavior provides a default implementation for send:to:with: that reflectively does the message send:

```
Behavior>>send: selector to: receiver with: arguments
   ^receiver perform: selector withArguments: arguments.
```

We can now override this method in the specific meta-class to change the semantics of message sends. For instance variable, we have to provide

two links: one for instance-variable read access, one for assignments.

The link for reading an instance variable is defined like this:

```
link := GPLink new metaObject: #class;
      selector: #iVarAt:In:
      arguments: #(offset object)
```

The other link for instance variable assignments:

```
link := GPLink new metaObject: #class;
      selector: #iVarAt:In:put:
      arguments: #(offset object newvalue)
```

Both the methods iVarAt:In: and iVarAt:In:Put: are provided as default method in class Behavior, we can override these methods on a concrete meta-class, for example to make the state of all objects of this class persistent.

Optimized links for sends. There are two optimizations possible: we should generate specific links for message sends with up to four arguments. A link for the case of zero arguments:

```
link := GPLink new metaObject: #class;
      selector: #send:to:
      arguments: #(selector object)
```

Specializing links for message sends helps to reduce the number of object allocations at runtime. In this case we do not need to create an array for the arguments.

Leveraging partial behavioral reflection. Forwarding message sends and instance variable accesses to the meta-class even when the default behavior is not overridden makes no sense. MetaClassTalk, for example, only compiles to a reflective call for those classes with changed behavior. We can realize this optimization by statically installing links only in those classes with overridden behavior. In all cases where the behavior is not changed, we use the default implementation provided by the virtual machine.

5.5 Implementation

Here we briefly discuss the implementation of partial behavioral reflection on top of the sub-method reflection framework, called GEPPETTO 2. The system is realized as an extension to PERSEPHONE, the open compiler framework described in Chapter 4 that is the basis of sub-method reflection.

5.5.1 The Transformation Plugin

As shown in Figure 5.6, we provide a new annotation, the LinkAnnotation. This is a non-textual annotation referencing the *link*. To add semantic meaning to this annotation, we provide a subclass of CompilerPlugin called GPTransfomer.

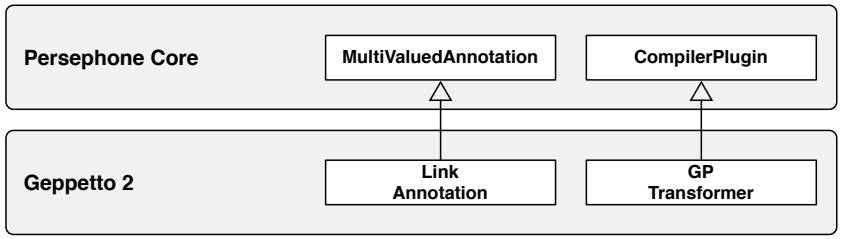

Figure 5.6: Extension for compile-time evaluation.

The GPTransformer transforms a copy of the AST so that the meta-object is called as specified in the link. Details can be found in the GPTransformer class of the REFLECTIVITY distribution[1].

5.5.2 Plugins for Reified Data

GEPPETTO supports data to be passed to the meta-object at runtime. Examples for this are the receiver and arguments from a message send, or the name of an instance variable. In addition, these reifications can be used as parameters for the condition or even as meta-objects. We discussed reified data in Section 5.3.2. A full list of all pre-defined reifications can be found in Table 5.1 on page 97.

We provide a plugin-architecture to add new kinds of reified data to be defined. All existing ones are implemented using this plugin architecture. All these so-called *parameters* are defined as classes, they share the common superclass GPParameter. To add a parameter, we add a subclass. We take GPObjectParameter as an example. The class needs two methods on the class side. First, a method key returning the name of the parameter:

```
GPNodeParameter class>>key
  ^#object
```

[1] http://www.iam.unibe.ch/~scg/Research/Reflectivity

The method nodes returns an array of all classes that can provide this data. The object is available in all node objects, thus we return an array containing the common superclass, RBProgramNode:

```
GPContextParameter class>>nodes
  ^{RBProgramNode}
```

The last method is responsible for code generation. It returns an AST which, when executed, results in the data we are interested in:

```
genForRBProgramNode
  ^RBVariableNode named: 'self'
```

With this plugin architecture, users of the framework can easily add reifications of data they are interested in. A typical example would be an analysis application for web-frameworks like Seaside. Here a very interesting piece of information is the currently executing session. With the help of a GPSessionParameter, we can reify this information and pass it for example to a condition of a link so that the analysis is scoped towards one particular session.

5.5.3 Code Quality

To show that we can generate more optimized code by leveraging the AST, we show a simple example. First we show how this is done on bytecode, then we discuss how the code can be optimized with the AST.

We have bytecode for the expression 3+4:

```
pushConstant: 3
pushConstant: 4
send: +
returnTop
```

We already discussed in Section 3.2.4 how to access runtime data like the receiver and arguments of a message send with bytecode transformation. For a *before* on a message send, we need to do the following:

1. We store the argument in a temporary variable.
2. We store the receiver in a temporary variable.
3. We call the meta-object.
4. We rebuild the stack so that the original message send can proceed normally.

The following examples shows the bytecode of the example expression 3+4 with the *hook* code added just before the original send of +:

```
pushConstant: 3          "original code"
pushConstant: 4          "original code"
popIntoTemp: 0           "put argument in temp 0"
popIntoTemp: 1           "put receiver in temp 1"
pushConstant: Logger     "push the meta-object"
pushTemp: 1              "push receiver for logging"
send: logReceiver:       "invoke the meta-object"
pop                      "end of meta-object call"
pushTemp: 1              "rebuild the stack"
pushTemp: 0
send: +                  "original code"
returnTop                "original code"
```

As can be already seen in this simple example, there is an overhead associated with this solution.

In the case of the AST, we can do better. An easy optimization can be done with just a little analysis of the node that we generate code for. If the receiver is a literal or a variable, we do not need to introduce temporary variables. In the case of the send seen in the example, the transformer has recognized the fact that the receiver is a literal and does not generate code to store the receiver in a temporary variable.

```
pushConstant: 3
pushConstant: 4
pushConstant: Logger
pushConstant: 3
send: logReceiver:
pop
send: +
returnTop
```

Another field where interesting optimizations are possible is cross link optimization when there are multiple links installed on one node. For example, when two links on the same message send are interested in the same data (*e.g.*, the arguments of a send as an array), it can be interesting to store this data once in a variable to be used by the code of both meta-object calls.

The current implementation has only some possible optimizations realized, we plan to improve optimized transformations, especially the case of cross link optimizations in the future.

5.6 Evaluation and Benchmarks

We first present benchmarks, then we discuss how our new model solves the problems we found with the bytecode-based solution (see Chapter 3).

5.6.1 Benchmarks

We presented previously in Chapter 3 a validation of *unanticipated partial behavioral reflection*. This validation has not been invalidated due to our new realization of top of our sub-method structural representation, as the new model is in principle the same. The only thing we need to make sure is that the performance has not degraded. To the contrary, we emphasized code quality (and thus execution speed) explicitly as one of the problems of the low-level representation used in the first implementation. Thus we need to show that our new approach is indeed *better* then the old. To show this, we compare our new system to the old one focussing on two key properties: performance of instrumentation and performance of instrumented code.

All benchmarks are run on an Apple MacBook Pro (2Ghz Intel Core Duo, 2GB RAM).

Performance of Instrumentation

We first compare GEPPETTO 1 and GEPPETTO 2 regarding instrumentation performance, which is the time it takes to install links on a piece of code before execution.

We use the package Network-IRC. This is an implementation of an IRC chat client. It consists of 39 classes, 751 methods and 2601 message sends that are not inlined in the bytecode. We install on these operations a very simple link: a message yourself send to the meta-object Object new.

To make both frameworks comparable, we only instrument those message sends that exist in the bytecode and thus can be annotated with both frameworks. As GEPPETTO 2 does on-demand code-generation, just comparing instrumentation time would not be entirely fair. The real transformation is done later on demand at runtime. Therefore, we modify GEPPETTO 2 to force code generation for every method where a link is installed.

Table 5.4 shows the result. When annotating without code generation, the new system is around 10 times faster. Even when forcing code generation for all possible methods (the worst case), GEPPETTO 2 still is two times faster.

In practice, we will not pay the price of full instrumentation at runtime,

System	time (msecs)	factor
Geppetto 1	8489	1.0
Geppetto 2	836	10.15
Geppetto 2 code gen	3880	2.19

Table 5.3: Instrumentation performance

as only a subset of methods actually gets executed. This will result in less time needed to generate code at runtime.

Execution Performance

To assess the performance of generated code, we use the example that we have shown in Section 5.5.3 to show that simple optimizations do lead to better runtime performance. The method #plus of class PlusB returns the result of the expression 3+4:

```
PlusB>>plus
        ^3+4
```

We install a link on the message #+:

```
link := GPLink new metaObject: Logger;
                selector: #logReceiver:;
                arguments: #(receiver).
```

The link requests the receiver to be passed to the meta-object as an arguments. We implemented this example in both GEPPETTO 1 and GEPPETTO 2. To assess the runtime, we execute:

```
b := PlusB new.
[10000000 timesRepeat: [b plus]] timeToRun
```

Running the above code with no links installed leads to a runtime of 1320 milliseconds. We need to deduct this from the overall runtime to get the actual time spend on the execution of the meta-object (code of the link and the execution of the empty method).

Table 5.4 shows the result. We record the base case of no installed links and both GEPPETTO versions. For overall runtime, the bytecode-based solution is slower by around 14 percent, whereas when we deduct the base case we can see that the actual execution of meta-object behavior is slower by 43 percent.

When we analyze the bytecode generated by GEPPETTO 1 and GEPPETTO 2, we see for one the effect of the optimization described in Section 5.5.3. We do not need to store the argument and the receiver in a temporary variable. The other optimization that helps with performance is link inlining. With GEPPETTO 1, the meta-object was never inlined into the bytecode but requested from the link and stored in a temporary variable. With GEPPETTO 2, we inline the meta-object into the bytecode as a literal object by default.

System	time (msecs)	factor	link (msec)	factor
No Links	1320		0	
Geppetto 2	1977	1.0	657	1.0
Geppetto 1	2252	1.14	932	1.42

Table 5.4: Comparing runtime performance

Of course, this a micro-benchmark showing pure link-call performance. In any normal setting, the meta-object will not consist of an empty method. Instead, code is executed by the meta-object. This code will take a large share of the overall runtime of a call to the meta-object, the call to the meta-object itself is only a tiny part of the overall execution time. Nevertheless, we have shown that the new system generates faster code than the original realization as presented in Chapter 3 based on BYTESURGEON.

5.6.2 Evaluation

At the beginning of this chapter, we mentioned three problems of the first realization of partial behavioral reflection (Chapter 3). In the following, we discuss how the system described in this chapter solves the problems.

Semantic Mismatch

The AST of sub-method reflection represents the structure of a method as a tree. All programming language concepts a programmer knows from the language are represented in this tree. Examples are control-structures: message sends like ifTrue: are optimized to jumps at the level of bytecode, but they exist as message sends in the AST. Blocks are another example: at the level of bytecode, block closures have no static representation, there is just code to create a block at runtime, which makes it hard to reason about the block-structure of code. The AST, however, represents blocks as RBBlockNode, including those that are part of control structures.

Code Quality

The stack model of bytecode complicates transformation at the level of bytecode. The problem is that at a certain bytecode instruction, we only know the layout of the stack, thus possible transformations are limited. To call a meta-object, we insert a so-called *hook*, a short piece of code that does the call to the meta-object. When we need to pass information to the meta-object, an additional *preamble* is needed that gets the needed information from the stack and stores it in a local variable.

The AST provided by sub-method reflection, in contrast, provides better support for transformations. The method structure is encoded as a tree, we can directly transform that tree and in many cases this means that we do not need to introduce additional variables or stack manipulation code.

Synthesized Code

When realizing behavioral reflection by transforming structure, we have the problem that this change is visible via introspection. For example, when adding a call to a tracer, at the level of the bytecode, this call is not distinguishable from base-level code. Introspection of the method, for example counting message sends, would lead to incorrect results. It is especially problematic for tools that work on a sub-method level like debuggers as they need to provide a mapping from the bytecode to the source code for the user.

This problem is not solved by performing the transformation at a higher level. Even when moving from bytecode towards the AST, the problem remains: transforming code for behavioral reflection destroys the original representation.

Nevertheless, sub-method reflection provides a solution to this problem. Sub-method reflection provides two levels of code: a high-level representation for reflection and a low-level representation for execution. The idea is to not transform the reflective model, but only the executed code which is never reflected on. This is realized by using the annotations combined with compiler plugins that give meaning (in the form of transformed code) to these annotations.

Performance and Extensibility

In principle the problems of semantic mismatch and code quality could have already been solved in Chapter 3 by not using bytecode as the basis for transformation. We could have used the AST as an intermediate representation. But there are reasons for using bytecode, as we have discussed

in Section 3.2.1:

Performance. Decompiling bytecode to the AST instead of a bytecode-near IR is costly.

No original language warranty. The bytecode we find in a method might not have been produced by a Smalltalk compiler, but some compiler for another language that compiles to Smalltalk bytecode.

Sub-method reflection solves both problems. The performance problem is solved by the persistent AST and on-demand code generation. The most expensive part of using an AST is decompilation of bytecode. With *sub-method reflection*, the AST is persistent, therefore we do not pay any cost for decompilation at all. In addition, we generate code on-demand. We therefore save time by not generating bytecode for methods that are not executed.

The problems of supporting other languages is solved by our PERSEPHONE compiler framework being an open implementation. We can, if needed, extend the AST to support other languages. The open compiler framework allows us to add new types of nodes for a non-Smalltalk language and provide code generation support easily.

5.7 Summary

We have presented a second realization of *partial behavioral reflection* where the links are annotations on the sub-method representation. We have simplified the overall framework considerably and extended it with support to reference reifications of method structure from the meta-objects. To show that our system can replace the bytecode-based model presented in Chapter 3, we have provided benchmarks. We have shown that our new system solves all problems identified in Chapter 3, in particular the problem of semantic mismatch, code quality and synthesized elements.

In the next chapter we discuss the remaining problem: we cannot apply behavioral reflection to an entire system.

Chapter 6

Modeling Meta-level Execution with Context

6.1 Introduction

We run into problems when applying behavioral reflection to either system classes or the code of meta-objects themselves, for example when applying a trace tool realized with behavioral reflection to itself. The reason for this is that calls to meta-objects will result in endless loops as soon as code is called from the meta-level that itself requests a call to the same meta-object. Behavioral reflection thus cannot be applied to the whole system.

First we present a simple example to illustrate the problem of *meta-object call recursion*. We then discuss the problem in detail and show that the cause for the problem is the missing representation of *meta-level execution*. The next section then provides an overview of *context* and *contextual reflection* which can solve the presented problem. We elaborate on an implementation, show benchmarks and after an overview of related work we conclude with a discussion of future work.

6.2 A Simple Example

We first provide a short overview of partial behavioral reflection and discuss how to implement a simple example, which we use in the rest of the paper.

The problem of meta-object call recursion is a known problem [29]. To show that it is relevant in practice, we have decided to keep our example

as simple as possible. In Section 6.6.2 we discuss as more complex scenario how our solution is useful for dynamic analysis in general.

Imagine that we want to trace system activity: we want the system to beep when it executes a certain method. This audio based debugging is an interesting technique to determine if a certain piece of code is executed or not. In Squeak, there is a class Beeper that provides all the beeping functionality. When calling beep, the beep sound is played via the audio subsystem. The following example shows how to create a link that will invoke the message beep on the Beeper class.

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep
```

Now we can install this beepLink on a method that is part of the system libraries. We take on purpose the method add: in OrderedCollection, a central class part of the collection libraries that is heavily used by the system. To set the link, we send the message link: to the AST-Node that stands for the whole method:

```
(OrderedCollection>>#add:) methodNode link: beepLink.
```

As result, a sound should be emitted each time the method OrderedCollection»#add: is called. But as soon as we install this link, the system freezes. This clearly was not the intended outcome.

6.3 Infinite Meta-object Call Recursion

Let's analyze the cause for the problem presented above. After a discussion of some ad-hoc solutions, we show that the problem is caused by a missing model for the concept of the *meta-level execution*.

The Problem. To ensure that the problem is not caused by our framework, we modify the example to call a different method at the meta-object, the method beepPrimitive which directly executes functionality defined in the virtual machine.

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beepPrimitive.
```

When installing this link, we can see that it works as expected: we can hear a beep for all calls to the add: method, for example when typing characters in the Squeak IDE.

The problem thus lies in the code executed by the meta-object. The Squeak sound subsystem uses the method add: of OrderedCollection at some

place to emit the beep sound. Thus, we execute the same method `add:` from the meta-object that triggered the call to the meta-object in the first place. Therefore we end up calling the meta-object again and again as shown in Figure 6.1. This is clearly not a suitable semantics for behavioral reflection: it should be possible to use behavioral reflection even on system libraries that *are used to implement the meta object functionality themselves*.

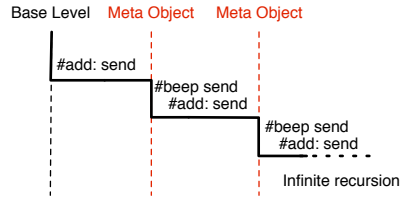

Figure 6.1: Infinite recursive meta-object call

We present now two ad-hoc solutions.

Code Duplication. As the problem is caused by calling base level code from the meta-object, one solution would be to never call base level code from the meta-object, but instead provide a renamed copy for all classes and use these from the meta-level. Duplicating the complete system has, of course, all the standard problems of code duplication: space is wasted. In addition, the copy can easily become out of sync with the original. The problems could be minimized by just copying those methods that are really needed. In practice, it is not easy to identify these methods, especially in dynamic languages. In addition this would cause changes in the reflective layer to become fragile because any change would require the programmer to update the copied version of the base level. This is clearly not a good solution.

Adding Special Tests. Another solution could be to add special code to check if a recursion happens. The problem with this solution is that it is ad-hoc, the code-base becomes cluttered with checking instructions. It just patches the symptoms, the recursive call, and does not address the real problem.

The Real Problem: Modeling Meta-level Execution. The ad-hoc solutions are not satisfactory. The real problem exemplified by a recursive meta-call is that when using meta-objects instead of infinite towers-of-interpresters, the awareness that an execution occurs at the meta-level has been lost. Normally activation of a meta-object implies a jump to the meta-level. The problem now is that this meta-level does not really exist in meta-object-

based architectures: there is no way to query the system to know whether we are at the meta-level or not.

It should be noted again that the problem we have seen is not specific to a particular behavioral reflection framework. We observe the same problem when applying MethodWrappers [24] to system classes. Method wrappers wrap a method with before/after behavior. MethodWrappers are reflectively implemented in Smalltalk and thus use lots of system library code during the execution of the wrapped methods. The same problem was identified for CLOS [29] and is thus present in other meta-class based systems like for example Neoclasstalk [20], or MetaClassTalk [17].

6.4 Solution: The MetaContext

We have seen that the real cause for the problem of endless recursion lies in the absence of a model for *meta-level execution*: the fact that the system is executing meta-level code is not represented.

6.4.1 Modeling Context

At any point in the execution of some piece of code we should be able to query whether we are executing at the meta or at the base level. Such a property can be nicely modeled with the concept of *context*: the *meta-level* is a context that is active or inactive.

With a way to model meta-level execution, it is possible to solve the problem of recursive meta-object calls. A call to the meta-object can be scoped towards the base level: a meta-call should only occur when we are executing at the base level. If we are already at the meta-level, the calls should be ignored. This way, meta-object calls are only triggered by the base level computation, not the meta-level computation itself, thereby eliminating the recursion.

We will first describe a simplified model that only provides two levels of execution (base and meta). We describe later how to extend our model to support multiple meta-levels.

The Metacontext. To model the meta-level, we introduce a special *context*, the MetaContext. This context is inactive for a normal execution of a program. MetaContext will be activated when we enter in the meta-level computation and deactivate when we leave it (Figure 6.2). The meta-context thus models *meta-level execution*.

A simple model with just one meta-context is enough to distinguish the meta from the base level. We will see later that it makes sense to extend the

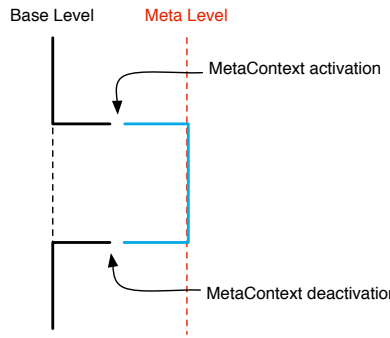

Figure 6.2: The MetaContext activation

meta-context to a possibly infinite tower of meta-contexts in Section 6.4.3.

Controlling meta-object activation. Just having a way to model meta-level execution via the meta-context is not enough to solve the problem of recursion, it is just the prerequisite to be able to detect it. We need to make sure that a call to the meta-object does not occur again if we are already executing at the meta-level. Thus, the call to the meta-level needs to be guarded so it is not executed if the execution is already occurring at the meta-level. In the context of behavioral partial reflection (*i.e.*, in the link-meta-object model that we used to show the problem), this means that the links are parameterized by the contexts in which they are active or not-active.

6.4.2 The Problem Revisited

With both the meta-context and the contextual controlled meta-object calls, we now can return to our example and see how our technique solves the problem of recursion. In our example, we defined the Beeper as a meta-object to be called when executing the add: method of OrderedCollection. The following steps occur (see Figure 6.3):

1. The add: method is executed from a base level program.
2. A call to the meta-object Beeper is requested:
   - We first check if we are at the meta-level. As we are not, we continue with the call.
   - We enable the MetaContext.
   - We call the meta-object.

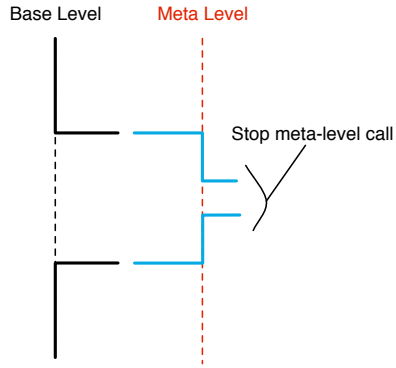

Figure 6.3: Stopping infinite meta-call recursion

3. Meta-object executes the beep method.
4. Meta-object calls the method add: method again
5. A Call to the meta-object is requested
    - We first check if we are at the meta-level. As we are executing meta-level code, the call is aborted.
6. Meta-object execution continues until it is finished.
7. On return to the base level, we deactivate the MetaContext.

Thus the recursive meta-call is aborted and the danger of recursion is eliminated. The model we described up to now with just one MetaContext is thus enough to solve the problem, but it not complete: it does not, for example, allow any calls to metameta-objects while already executing at the meta-level, which would make it impossible to observe or reason about metabehavior. In the next section, we therefore extend the model.

6.4.3 The Contextual Tower

As with the tower of interpreters, we can generalize the meta-context to form an infinite tower of meta-contexts. With the infinite tower of reflective interpreters, a reification is always bound to a specific interpreter. Normally, a jump from the base to the meta level means executing a reflective function that is defined as part of the interpreter I_1. But it is possible to define a reflective function one level up: this then is only triggered by the interpreter I_2 that interprets I_1, thus allowing us to reflect on the interpreter

I_1 itself. Figure 6.4 shows the reflective tower as visualized in the work of Smith [119].

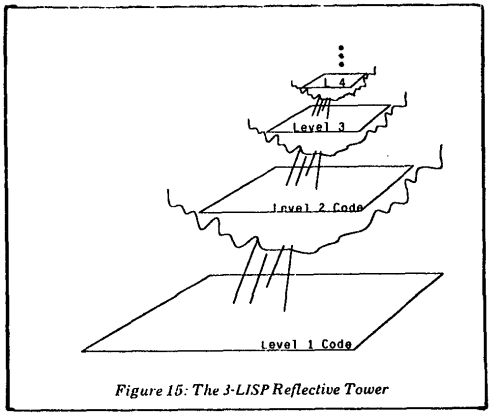

Figure 6.4: The 3-Lisp reflective tower from [119]

Transposed to our contextual model, it follows that having just one context (the meta-context) is not enough. We need more contexts for meta-meta, meta-3 and so on. If we have this *contextual tower*, we can for example define a meta-meta object that is only called when we are executing at meta-1. Meta-object calls need thus not only be defined to be active for the base level, but they can optionally be defined to be active for any of the meta-levels. This allows us to define meta*objects that reason about the system executing at any level, as with the endless tower of interpreters.

As with all infinite structures, the most important question is how to realize it in practice. For the case of the infinite meta-context tower, there is an easy solution: contexts are objects, so they can have state. We can parameterize the meta-context object with a number describing the meta-level that it encodes. Shifting to the meta-level means shifting from a context n to a context $n + 1$.

Figure 6.5 shows an example. We have three contexts: the base level, the meta-level and meta-2. We see two links that are active at the base level. When called, they activate the meta-context with level 1. Then in the code of the meta-object (either the code of the meta-object itself or any library code executed), we have a third link. This link is defined to be active only on meta-level 1, thus it will on execution enable meta-2. We show an example of such a level 1 link in Section 6.6.2.

An interesting and very nice property of the parameterized context is that the meta-contexts are only created on demand, they do not exist

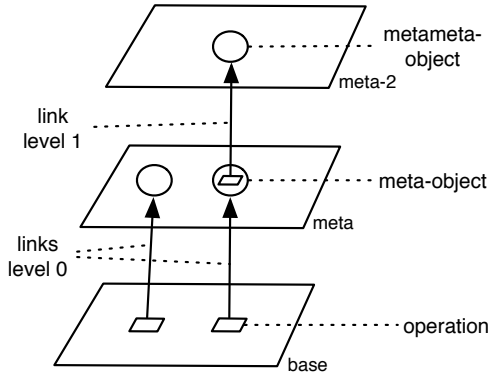

Figure 6.5: The Contextual Tower up to the second level

as an endless tower. This means, we have a form of partial reflection for providing a potential endless tower. If a meta-level is not needed, it does not cost anything.

In the following, we extend our simple context representation to support such an endless tower of contexts.

6.4.4 MetaContext Revised

The simple version of our idea with just one MetaContext is not enough to allow us to encode the Contextual Tower. We need a slightly modified model of MetaContext where the MetaContext is parameterized by the *level*. Such a parameterized MetaContext is not simply active or inactive, it is active for the level that it currently is set to. Thus when querying such a MetaContext, we give as a parameter a number denoting a meta-level. We will in the next section see how to realize such a context in practice.

6.5 Implementation

Now that we have described the solution in general, we present an implementation of that model for REFLECTIVITY, our reflection framework. We first show how the context is implemented and then discuss contextual links.

6.5.1 Implementation of MetaContext

The MetaContext is a class that has one instance per thread (a thread-specific singleton). The instances are created on demand and are stored per thread. As threads are objects in Squeak, we extended them to be able to store additional state directly in an associated dictionary. This mechanism is then used to store the MetaContext instance.

Querying context. The MetaContext needs to model the meta-level. For that, it has one instance variable named level. We can increase the level by calling shiftLevelUp or decrease by shiftLevelDown. To test if the MetaContext is active for a certain level n we can call isActive: with a parameter denoting the level:

```
MetaContext current isActive: 0
```

Sending current to the class MetaContext will retrieve the MetaContext singleton from the current process. If there is none yet, it will lazily create a MetaContext with the level set to 0. Thus for a normal base level execution of code the expression above will return true.

Executing code in the MetaContext. To change the meta-level that code is executed at, we provide a way to run a block (an anonymous higher order function) one meta-level higher than the code outside the block. For example, to execute at meta-1, evaluate:

```
[ self assert: (MetaContext current isActive: 1) ] valueWithMetaContext
```

If code is already executing at meta-1, calling valueWithMetaContext again will execute at meta-2:

```
[[ self assert: (MetaContext current isActive: 2)
    ] valueWithMetaContext] valueWithMetaContext
```

The method valueWithMetaContext is implemented in the BlockClosure object. It will first shift the level of the current MetaContext up, then the block is executed. At the end the level of the context is shifted down to the previous value. We make sure that the downshift happens even in case of abnormal termination by evaluating the block using the exception handling mechanisms of Smalltalk:

```
valueWithMetaContext
  MetaContext current shiftLevelUp.
  ^self ctxtEnsure: [MetaContext current shiftLevelDown]
```

To make sure that the execution of the context handling code itself does not result in endless loops, we do not call any code from system libraries in

this method. Instead, we carefully copy all methods executed by the context setup code. The copied methods reside in the classes of the system library, but they are prefixed with *ctxt* and edited to call only prefixed methods. One example is the call to ctxtEnsure: seen in the method above.

Concurrent meta-objects. As the MetaContext is represented by a thread-specific singleton, forking a new thread from the meta-level would mean that this thread has its own MetaContext object associated which is initialized to be at level 0. We have solved this problem by changing the implementation of threads to actually copy the meta-level information to the newly created thread. A thread created at the meta-level thus continues to run at the meta-level:

```
[ [ self assert: (MetaContext current isActive: 1) ]  fork ] valueWithMetaContext
```

As the context information is not shared with the parent thread, the meta-level of the new thread is independent. It can continue to run at the meta-level even when the parent thread has already returned to the base level.

6.5.2 Realizing Contextual Links

Now that we have a suitable way to represent the meta context, we need to make the links and the code that is generated to call them *context-aware*. For that, we need to solve three problems:

1. the link needs to be defined to be specific to a certain meta level.
2. link activation should occur only when code is executing at the right meta-level.
3. link activation should increase the meta-level.

The first problem is solved by a simple extension of the Link class, whereas the other two are concerned with the code that our system generates for link activation. We will now show the required changes in detail.

Meta-level Specific Links. To allow the programmer to specify that a link is specific to a certain meta-level, we extend the link with a parameter called *level*. If *level* is not set, the link is globally active over all links (the standard behavior). The level can be set to any integer to define a link to only be active at that specific meta level. For our example, a link that is active only when executing base level code looks like this:

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep;
```

```
beepLink level: 0.
```

Context-Aware Link Activation. To jump up one meta-level on link activation, we make sure that the code generated for a link is wrapped in a send of valueWithMetaContext. The resulting code will look like this:

```
[ ... code of the link  ... ] valueWithMetaContext
```

In addition to that, we need to make sure that the link is only called when we are on the correct meta level. This is done by checking if the current MetaContext is executing on the same level as the level that the link is defined to be active. Only if this is true, we activate the link and call the meta-object. The code we need to generate looks like this:

```
(MetaLevel current isActive: link level)  ifTrue: [
    [ ... code of the link  ... ] valueWithMetaContext
].
```

We will not go into the detail of how exactly the code is generated. The code can be found in the class GPTransformer of the REFLECTIVITY distribution[1].

6.6 Evaluation and Benchmarks

We first show that our solution solves the recursion problem, then we discuss how meta-context is useful for dynamic analysis. We describe *dynamic meta-level analysis* and realize an example. We present benchmarks to show the practicability of our approach.

6.6.1 The Problem is Solved

We show that we can really solve the practical problem. For that, we define the link that activates the Beeper to be specific to level 0:

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep;
beepLink level: 0.
```

Now we can install the link:

```
(OrderedCollection>>#add:) methodNode link: beepLink.
```

[1] http://www.iam.unibe.ch/~scg/Research/Reflectivity

As soon as the link is installed, the next call to the `add:` method will trigger code-generation for that method. The code-generator will take the link into account and generate code as described earlier. Thus the recursive call to the `add:` method will not occur. We will thus hear a beep for every call to the `add:` method from the base level only.

6.6.2 Benefits for Dynamic Analysis

Our initial reason for modeling meta-level execution was to solve the problem of meta-object call recursion, and thus making reflection easier to use. But the solution we presented, modeling meta-level execution with the help of a meta-context, is useful far beyond just solving the problem of recursion. In this section, we will discuss what it means for dynamic analysis. We show how it allows the programmer to analyze the code executed by meta-objects by enabling *dynamic meta-level analysis*.

Dynamic Analysis. One of the applications for behavioral reflection is dynamic analysis. For example, with reflection, it is fairly easy to introduce tracers or profilers into a running program [42] that normally require changes at the level of the virtual machine. One problem, though, with using reflection to introduce analysis code is that it is not clear which of the recorded events are resulting from the base level program execution and which from the code of the tracer itself. As long as we only trace application code, we can easily restrict the reflective tracer to the code of the application. But as soon as we want a complete system trace, we start to get into problems: recursion can occur easily (the problem we solved earlier), but even after working around recursion, we face another problem: how do we distinguish events that originated from the application from those that only occur due to the code executed by the tracer itself?

With our meta-level execution context, the problem described does not occur at all. The tracer (or any other tool performing dynamic analysis) is actually a meta-level program. A simple tracer would be the meta-object itself. More complex analysis tools would be called from a meta-object reifying the runtime event we are interested in. Thus, the code that performs the dynamic analysis is executing at the meta-level, while the links that trigger it are only active when executing a base level computation. This way we make sure that the infrastructure performing our analysis never affects the trace itself.

Dynamic Meta-level Analysis. An interesting challenge for dynamic analysis is that of analyzing the meta-level execution itself. Meta-level code should be lean and fast to not slow down the base computation more then really necessary. We thus are very interested in both tracing and profiling meta-level execution.

Our explicitly modeled meta-level execution makes this easy: we can define a link to be only active when executing at the meta-level. Therefore, we can install for example a trace-tool that exactly provides a trace of only those methods executed by a meta-object, even though the same methods are are called by the base level at the same time. We can thus as easily restrict the tracer towards meta-level execution as we restrict it to trace base level programs.

For our example, this means that we can use dynamic meta-level analysis to find the place in the sound-subsystem where the recursion problem happens when not using contextual links. In Section 6.3 we discuss that recursion happens, but we do not know where exactly the recursive call happens.

We define a link that is only active when we are executing code at level 1:

```
loggerLink := GPLink new metaObject: logger;
        selector: #log:;
        arguments: #(context);
        level: 1.
```

This link sends the message log: to the logger. The logger is an instance of class MyLogger:

```
logger := MyLogger new.
```

The method log: records the stack-frame that called the method where the link is installed on:

```
MyLogger>>log: aContext
   contexts add: aContext home sender copy
```

We install both a link calling the Beeper that is specific to level 0 and our link that is specific to level 1 on the method add:.

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep;
beepLink level: 0.

(OrderedCollection>>#add:) methodNode link: beepLink.
(OrderedCollection>>#add:) reflectiveMethod methodNode link: loggerLink.
```

We can now inspect the logger object and see that it is recording the execution of SoundPlayer class»startPlayingImmediately: for every beep. Looking at this method, we find the code ActiveSounds add: aSound., which is the one spot in the sound system that calls the method add: of OrderedCollection. Thus

we found with the help of dynamic meta-level analysis the exact call that causes the recursion problem as shown in Section 6.2.

6.6.3 Benchmarks

To assess if the system as presented is practically usable, we have carried out some benchmarks. Without the additional code for context activation, there is no overhead at all for calling a meta-object besides the call itself. The link specifies the meta-object and which method to call. The system generates from that information code that just calls the meta as specified. The problem now for analyzing the new context-enabled system is that the context code will, compared to the standard link activation, take a lot of time. In practice, though, meta-objects are usually there to do something: there is always code executed at the meta-level. So to make a practical comparison of the additional percentage of slowdown introduced by the context code, we need to compare the context-setup code not only to the link activation of an empty meta-object, but to a meta-object that actually executes code.

We will benchmark the slowdown of the context handling for the execution of different meta-objects. The variation between the meta-objects is the number of sends done at the meta-level. For the benchmark, we create a class Base with just one empty method bench. To play the role of a meta-object, we create a class Meta with one method in which we call an empty method in a loop. This method thus simulates a meta-object doing some work. We can easily change the amount of work done by changing the loop. To know how this simple benchmark compares to real code executed, we added a meta-object calling the Beeper and one converting a float to a string.

We install a link on the method in Base to call the method in Meta. We call the base method now in a loop and measure the time:

```
[100000 timesRepeat: [Base new bench ]] timeToRun
```

Table 6.1 shows the result when comparing both the original GEPPETTO and the context-enabled GEPPETTO for different meta-objects[2]:

As expected, the slowdown in the case of an empty meta-object is substantial. But as soon as the meta-object itself executes code, the overhead starts to be acceptable. For calling the Beeper (from our running example), we have found an overhead of 63%. We are down to 17% on when executing 500 empty methods, and at 6.4% on 1000 methods.

[2]The benchmark was run on an Apple MacBook Pro, 2.4Ghz Intel Core 2 Duo with 2GB RAM on Squeak Version 3.9

meta-object	context (msecs)	standard (msecs)	slowdown
0 message sends	614	34	1'705.88%
10 message sends	723	165	338.18%
50 message sends	1040	470	121.28%
Beeper	1543	942	63.80%
100 message sends	1406	856	64.25%
200 message sends	2236	1621	37.94%
1234.345 printString	2534	1920	31.98%
500 message sends	4580	3907	17.23%
1000 message sends	8543	8029	6.40%

Table 6.1: Slowdown of meta-object calls with context

It should be noted that this does not mean that the overhead observed for meta-object calls translate directly into a slowdown of a program using reflection. In a real program, the overall slowdown depends on how often meta-objects are called and how much time the program spends at the base level and meta-level compared to switching meta-levels.

6.7 Related Work

Context-Oriented Programming. ContextL [33, 78] is a language to support *Context-Oriented Programming* (COP). The language provides a notion of *layers*, which package context-dependent behavioral variations. In practice, the variations consist of method definitions, mixins and *before* and *after* specifications. COP has no first-class notion of *context*, it is implicitly defined by the layers that are activated. The topic of reflection has been discussed for COP [34]. But the topic discussed is reflective layer activation, not a reflective model of context or the use of context to structure or control reflection itself.

Aspects. The concept of context has seen some use in AOP [127]. As context specific behavior tends to crosscut base programs, it can be implemented advantageously as aspects. This leads to the notion of context-aware aspects, *i.e.*, aspects whose behavior depends on context. The work has been continued in the direction of supporting reasoning on contexts and context history on the level of the pointcuts [75, 76].

Deployment strategies [126] provide full control over dynamic aspect deployment both related to the call stack and to created objects and func-

tions. AspectBoxes [13] is another example where aspects are controlled via a context, in this case defined by the classbox model [10]. A good overview and discussion on the many mechanisms for dynamically scoping crosscutting in general is described in the work of Tanter [125].

Stratified Aspects [15] define an extension to AOP that identify spurious recursion to be a problem and present the concept of meta-advice and meta-aspects as a solution. This idea has some similarities to contextual reifications, with the exception that meta-level execution is not modeled explicitly.

The MetaHelix. The problem of unwanted meta-level call recursion has been mentioned by Chiba and Kiczales [29]. The problem discussed is first the structural problem that *e.g.*, fields added by reflection to implement changed behavior show through to any user of introspection. The other problem mentioned is recursion, as any use of changed behavior can trigger a reification again. As a solution, the authors present the *MetaHelix*. All meta-objects have a field implemented-by that points to a version of the code that is not reflectively changed.

This approach is both more general and restrictive than our context based solution. It is more general, as it tries to solve the problem of the visibility of structural change. And it is more restrictive, as it does not model meta-level execution. The programmer has to call the right code explicitly, thus it can be seen as a controlled way to support the code copying solution presented in Section 6.3. The problem of structural changes is a very interesting one. As future work we plan to apply the ideas of the meta-context to structural reflection.

Subjective Programming. Us [122] is a system based on Self that supports subject-oriented programming [73]. Message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContexL. Thus subjectivity is applied to the problem of controlling the access to reflection APIs, subjectivity is not used for controlling behavioral reflection.

6.8 Summary

In this chapter we have analyzed the problem of the missing representation of meta-level execution in meta-object architectures. We have shown that the problem of infinite meta-object call recursion can be solved by introducing a representation for meta-level execution. We proposed to model the execution at the meta-level as a first class context and presented an implementation. Benchmarks show that the implementation can be realized in a practical manner.

With this extension, the last of the three problems we noted in Chapter 2 is fixed. We have extended structure to cover sub-method elements, added support for behavioral reflection using annotations and made sure that behavioral reflection can be used everywhere in the system, including system classes and meta-object code itself.

In the next chapter, we show as a case study how to use the resulting complete system, named REFLECTIVITY, in practice. We take *dynamic feature analysis* as an example.

Chapter 7

Case Study: Dynamic Feature Annotation

7.1 Introduction

In this chapter, we present a use-case for our REFLECTIVITY system. We discuss how to use annotation-based partial behavioral reflection in the context of dynamic analysis. First we give a brief introduction to feature analysis and the problem of trace-based approaches. Then we present feature annotation as a solution: instead of recording complete traces, we directly annotate at runtime the static *sub-method* structure with information about features.

The goal of this chapter is to show that the annotatable static model of the system provided by sub-method reflection is not only useful as the implementation basis of behavioral reflection. In addition, we can, at runtime, access the static structure from the behavioral meta-objects and annotate it with dynamic information. The annotations are then available to tools, for example for visualization.

7.2 Dynamic Feature Analysis

We give a brief introduction to dynamic feature analysis before we discuss the problems with trace-based approaches that we solve with feature annotation.

7.2.1 Feature Analysis in a Nutshell

Traditionally, reverse engineering techniques focused on analyzing source code of a system [31]. In recent years, researchers have recognized the significance of centering reverse engineering activities around the behavior of a system, in particular, around features [55, 90, 118]. Reasoning about object-oriented systems in terms of features is difficult, as they are not explicitly represented in the source code. The first step therefore is to define what is meant by a feature, establish a feature representation and to locate the relevant parts of the source code that participate in its behavior.

The goal of feature analysis is to reason about a system in terms of its features. A fundamental step of any feature analysis approach is to first apply a feature identification technique to locate features in source code. As a basis for feature analysis we use a model which expresses features as first class entities and their relationships to the source entities that implement their behavior [63]. Once the representation of a feature is established, we can reason about a system in terms of its features. Furthermore, we can enrich the static source code perspective with knowledge of the roles of classes and methods in the set of modeled features.

The generally adopted definition of a feature is a unit of observable behavior of a system triggered by a user [2, 55, 89, 132, 133]. Techniques for feature identification through dynamic analysis typically instrument a system, capture traces of feature behavior and establish links to source code. However, capturing dynamic data to represent features raises many issues that need to be taken into consideration.

7.2.2 Problems

Large Amounts of Data. The volume of trace data generated represents a threat to the scalability of any feature analysis approach. As the granularity required for an experiment increases, so too does the volume of information generated.

Dynamic analysis approaches adopt different strategies to deal with large amounts of data. Some of the most popular strategies adopted by researchers to tackle and analyze dynamic data are: (1) summarization through metrics [49], (2) filtering and clustering techniques [71, 135], (3) visualization [32, 65] (4) selective instrumentation and (5) query-based approaches [112]. Many techniques apply a combination of these strategies.

Fine-grained Analysis. Traditionally, dynamic analysis techniques for feature analysis focused on execution traces consisting of a sequence

of method executions [55, 133]. Some dynamic analysis approaches trace additional properties of behavior such as the message receiver and arguments or instance creation events [36, 66]. Most existing feature analysis techniques capture traces of method events but they do not capture behavioral data of sub-method elements such as variable assignments [118, 90].

However very little work in feature analysis has focused on a means to model which sub-method entities are part of a feature.

Link to Source-code. Trace-based techniques result in large amounts of data which are extracted for post-mortem analysis. As a result it is difficult to maintain a link between the behavioral data corresponding to a feature and the source code. We would like to embed the high-level knowledge of features directly in the source code and perform feature analysis in this context. Unfortunately, the text-based format of source code does not facilitate such needs.

Variations. Moreover the definition of what behavior constitutes a feature is not clear. Questions like how many paths of execution and how much variation of inputs arise. Ideally we would like to *grow* features over a series of runs of a system. We capture different variations of a user scenario or feature and attribute the resulting behavior to the same feature.

Performance. Gathering traces at runtime always slows the system down considerably. The additional code inserted is costly. In addition, creating the trace data-structure will put pressure on the memory management.

7.2.3 Summary

We can see the following problems with trace-based dynamic feature analysis approaches:

- Dynamic feature analysis implies a need to manipulate large amounts of trace data.
- Current feature analysis techniques do not consider analysis to the granularity of sub-method elements (*i.e.*, variable assignments).
- It is hard to support multiple runs to grow features (to take different execution paths into account).
- There is no easy way to embed information about features in the source code.

- Trace-based approaches slow down the execution of the system.

7.3 Feature Annotation

Feature identification (*i.e.*, locating which parts of the code implement a feature) by dynamic analysis is done at runtime: the feature is executed and the execution path is recorded. For example, when exercising *Login* feature of an application, we record all methods that are called as a result of triggering this feature. This trace of called methods then encompasses exactly all those methods that are part of the login feature.

7.3.1 Dynamic Analysis with REFLECTIVITY

Trace-based feature analysis can be easily implemented using partial behavioral reflection. In a standard trace-based system, the tracer is the object responsible for recording the feature trace. This tracer is the meta-object (see Figure 7.1). We define a link that calls this meta-object with the desired information passed as a parameter (*e.g.*, the name and class of the executed method). The link then is installed on the part of the system that we want to analyze. When we then exercise the feature, the trace meta-object will record a trace.

The resulting system is very similar to existing trace-based systems, with one exception: tracing now can easily cover sub-method elements, if required.

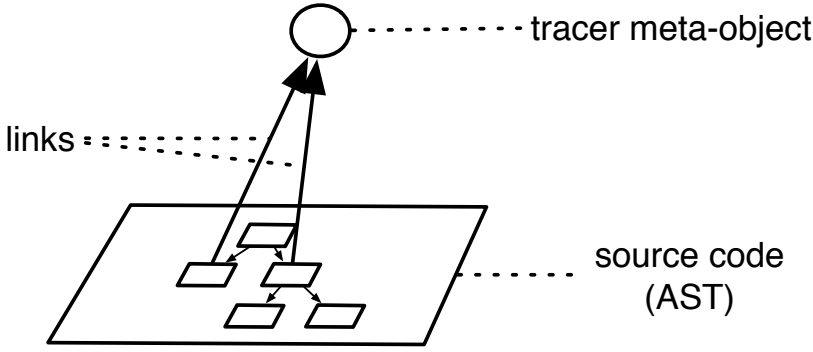

Figure 7.1: A tracer realized with partial behavioral reflection.

7.3.2 Feature Annotation with REFLECTIVITY

In contrast to traditional dynamic feature analysis approaches, our sub-method reflection based approach does not need to retain a trace. The goal of feature identification is to map features to the source code. With the annotatable representation provided by sub-method reflection, we can annotate every statement that participates in the behavior of a feature. Instead of recording traces, we tag all the AST nodes that are executed as part of a feature with a feature annotation at runtime.

We install the link on all the AST nodes of the system that we plan to analyze. Exercising the feature subsequently annotates all methods or instructions that take part in a feature execution. In this way we do not need to retain traces, resulting in less data to be managed.

7.3.3 Deinstallation at Runtime

Performance is a major issue when using *dynamic analysis*. Our goal is therefore to limit both where and when we use behavioral reflection. The *where* we can control by installing the link only on those places in the system that we are interested in analyzing. For the *when*, we leverage the fact that we can deinstall links at runtime from the meta-object.

When we tag a node to be part of a feature for the first time, there is no need to retain the link to the tagging meta-object. Any subsequent call as part of the feature just would set the annotation again. A call that is not part of the feature would have no influence on the feature annotation, therefore, we can remove the link after annotating the node. This removal will result in new code being generated on the next call of the method. This generated code then does not call the meta-object on all those nodes that already are tagged to be part of the feature, resulting in less slowdown of the application.

7.3.4 Growing Features

Our feature annotation approach can easily support many of the existing feature analysis approaches. For example, we could exercise a feature multiple times with different parameters to obtain multiple paths of execution. This can be important, as the traces obtained can very considerably depending on the input data.

For trace-based approaches this results in a many-to-one mapping between features and traces. Using our approach, if the execution path differs over multiple runs, newly executed instructions will be tagged in addition

to those already tagged. Thus we can use our approach to iteratively build up the representation of a feature covering multiple paths of execution.

The link-condition is inlined into the bytecode of the method. The systems detects the special case when the condition is statically true or false and optimizes the generated code. When it is set to be false, our framework will not generate any code for the link, not even a call to the condition itself. The code generated is the same as if the link would be de-installed. This makes it possible to let the tagging link be installed after a first feature analysis. We can at any point in the future enable the link again if we want to do an additional run to grow the feature.

7.3.5 Implementation

In the following, we describe the implementation of feature annotation using REFLECTIVITY. The following classes have been implemented:

FeatureAnnotation A subclass for defining the annotation #feature. The annotation references the symbol describing the features.

FeatureTagger This is the meta-object. Its main responsibility is to annotate the node for which it is set as a meta-object. In addition, it creates the link that is used to call it.

FeatureController This class provides a simple user interface. It supports to define a name and to stop and restart feature annotation. The controller is instantiated for the package to be analyzed.

We now give a brief overview of the implementation of the FeatureTagger, the link and the FeatureController.

FeatureTagger. The FeatureTagger is the meta-object that we call to annotate a node. The name of the feature that it annotates is stored in an instance variable. The message sent is tagNode:link: with both the node and the link as parameters. With these, it sets the annotation and removes the link:

```
FeatureTagger>>tagNode: aNode link: link
  aNode addFeatureAnnotation: feature.
  aNode removeLink: link.
```

Link. The link is created when instantiating a new feature tagger:

```
FeatureTagger class>>linkFor: aFeature
  ^GPLink new
```

```
    metaObject: (self newFor: aFeature);
    selector: #tagNode:link:;
    arguments: #(node link).
```

The FeatureTagger is created and managed by the class FeatureController.

FeatureController. We provide a very basic way to control the feature tagger and its link. The class FeatureController provides a basic command interface. This interface supports to create a controller for a feature, to set the package to analyze, to install the link as well as starting and stopping feature analysis itself:

```
controller := FeatureController for: #open
controller package: 'Network-IRC'.
controller installLinks.
controller featureStart.
controller featureStop.
```

An interesting implementation aspect is how the FeatureController manages link activation. The control attribute of the link is set to the instance variable of the FeatureController:

```
FeatureController>>taggerLink
  ^link ifNil: [
    link := FeatureTagger linkFor: feature.
    link condition: [active]].
```

As we reference the instance variable in a block, we can enable or disable the links by setting the instance variable active of the controller.

In addition, when we plan to disable feature analysis for a longer period but plan to grow the feature later, we can set the condition to false:

```
FeatureController>>deactivate
  link condition: false.
```

This will result in all methods being recompiled and the link calling code being completely removed. The links are retained, though, as annotations. As soon as we want to continue analyzing for this feature, we can activate the controller again:

```
FeatureController>>activate
  link condition: [active].
```

7.4 Validation

We validate feature annotations on the one hand by presenting benchmarks to show that the deinstallation directly after annotation results in far superior performance. On the other hand, we present an experiment to show the difference of the amount data gathered between a trace-based approach and the annotation based approach. We use the content-management system Pier [110] as a case-study.

7.4.1 Benchmarks

The goal of the benchmark is to show that de-installing links at runtime saves time. Pier has an extensive test-suite. We install the feature annotations system on the Pier classes that are in the sub-package Pier-Model (59 classes, 418 methods). We analyze method granularity:

```
ui := FeatureController for: #test
ui package: 'Pier−Model'.
ui installLinks.
ui featureStart.
```

Then we run the tests of Pier three times (see Table 7.1):

No feature annotation. We execute the tests without feature annotation as the base case.

Feature annotation remove. Feature tagging is enabled, the link is removed after setting the annotation.

Feature annotation no remove. We use a modified version of the feature tagging code where the link is not removed.

	time (msecs)	factor
No feature annotation	1897	1.0
Feature annotation remove	3787	2.00
Feature annotation no remove	7290	3.84

Table 7.1: Performance of Annotation

We can see that deinstalling the link improves performance, even though the bytecode needs to be regenerated for all the methods where links are removed.

7.4.2 Number of Events Generated

To assess the possible space saving due to annotating the static structure as opposed to recording traces, we show the difference in events generated in both cases. To measure the size of a trace, we install a counter that records method invocations while exercising a feature. When we annotate features, the result are annotations on the static structure. Therefore, we count the resulting annotations.

The program used is again the content-management system Pier. We only annotate the package Pier-Model (59 classes, 418 methods).

Table 7.2 shows both numbers for different features:

Feature	Number of events	Number of annotations	Factor
Display Page	2655	150	17.7
Call Page Editor	2797	168	16.65
Save Page	3020	222	13.60
Page Settings	2515	143	17.59

Table 7.2: Dynamic events compared to number of annotations

The number of dynamic events that a tracer would record is thus far larger than the resulting entities annotated with feature annotation.

7.4.3 Evaluation

We revisit the problems identified at the beginning of the chapter.

Large Amounts of Data. Due to annotating features instead of recording complete traces, we reduce the amount of data.

Fine-grained Analysis. *Sub-Method Reflection* provides support for sub-method abstractions.

Variations We can grow features, as we can control the tagging links.

Link to Source-code. Annotations of sub-method structure can be read from any tool for analysis and visualization.

Performance. As we can deinstall the code after the first call, we do not slow down the execution of the complete run.

7.5 Summary

In this chapter we have presented dynamic feature annotation, a technique that solves some issues found in traditional trace-based dynamic feature analysis approaches. Feature annotation support the analysis on a sub-method level and does not require to store complete trace data.

With feature annotation we have presented an example of the interplay of *sub-method structural reflection* and *behavioral reflection*. The structure is used for on the one hand as the basis for behavioral reflection, on the other hand we use the structural model for storing the result, *i.e.*, which part of the system is used by which feature. The annotated sub-method structure then can be used directly as an input for other tools or to visualize features.

In the next chapter, we conclude the dissertation by reviewing contributions and impact of the work presented and we discuss future work.

Chapter 8

Conclusions

In this chapter we briefly summarize the contributions of the thesis, discuss the impact and possible future work.

8.1 Contributions of the Dissertation.

The contributions of the thesis are:

Unanticipated partial behavioral reflection. We realized *partial behavioral reflection* in the context of a dynamic language so that it can be deployed at runtime.

Sub Method Reflection. We have extended the standard model of structural reflection to cover sub-method elements.

Annotation based partial behavioral Reflection. We have presented how partial behavioral reflection can be realized on top of sub-method structural reflection.

Contextual Reification. We extended partial behavioral reflection with the concept of *context* and have shown how this solves the problem of infinite meta object call recursion.

Dynamic Analysis and Feature Annotations. We discussed the use of our solution for dynamic analysis. In detail we presented a case-study of dynamic feature analysis using *feature annotations*.

Implementation. Both sub-method reflection and unanticipated partial behavioral reflection have been implemented in Squeak Smalltalk. We have presented the implementation in detail and shown its practicability by realizing examples and presenting benchmarks.

8.2 Impact

REFLECTIVITY has already seen some use in research, both at the University of Bern and other research groups. We briefly survey the relevant projects. Publications are cited where they already exist.

HistOOry. Frédéric Pluquet (deComp, Universite Libre de Bruxelles). The goal of the project is to experiment with partial persistence following the fat node model as proposed by Tarjan [47] in the context of object oriented languages. A first paper [107] outlines more information on the general goal the project. The results shown in this paper are obtained an early Java based prototype. A new implementation of partial persistence has been build using GEPPETTO.

Transactional Memory. Lukas Renggli (Software Composition Group, University of Bern) has used Reflectivity for an experimental implementation of *Software Transactional Memory* [111]. It was realized with no change to the virtual machine using the system as presented in this thesis.

Freezable Traits. A prototype of *freezable traits* [53] uses BYTESURGEON. Traits provide groups of methods that can be composed with other traits and used by classes. One problem with traits is the resolution of naming conflicts: if two traits provide methods with the same name, we need to select which one to use. Freezable traits provide an expressive composition mechanism to solve such conflicts.

Object Flow Analysis. Adrian Lienhard (Software Composition Group, University of Bern). TREENURSE was used in an experiment that analyzes how objects flow through an object-oriented program at runtime [95, 93, 94]. The work was done prior to the existence of a full implementation of partial behavioral reflection. Despite this, the less powerful inlining model provided by TREENURSE proved to be very valuable.

Back-In-Time Debugger. Chrisoph Hofer used BYTESURGEON to realize an omniscient debugger [80, 79].

Pluggable Types. Nik Haldiman used annotations on sub-method structure to implement *pluggable types* [67, 68, 69].

Test Coverage Analysis. Stefan Reichhart (Software Composition Group, University of Bern). The TREENURSE instrumentation framework has been used to implement a code coverage tool that provides line-by-line coverage analysis. Code coverage then was the basis for a tool to asses test coverage of a system and an automatic test smell analyzer [109].

Dynamic IDEs. David Röthlisberger (Software Composition Group, University of Bern) is working on development environment that show dynamic information as part of the traditional view of classes and methods to

support code understanding [117].

Aspects. Ongoing work at the Software Composition Group aims at realizing AOP on top of partial behavioral reflection as pioneered by Reflex [127]. The goal is to explore advanced scoping of aspects to continue of our earlier work on *Context Aware Aspects* [127].

8.3 Future Work

We now discuss future work. Initially we discuss work related to the sub-method structural representation and subsequently those specific to partial behavioral reflection.

Sub-method Structural Reflection

Compactness of the representation is an interesting field for future work. For our experiments, memory consumption has never posed a major problem. Nevertheless, we plan to research how to improve storage by optimizing the AST representation leveraging transparent AST compression [58].

Our current implementation of the annotation framework could be extended to support not only sub-method elements, but all structural entities like classes and packages.

Another interesting research direction is to consider the use of *reflective methods* to replace the text-based storage of source code. The human-readable code could be reconstructed from the reflective representation instead. The current version provides some support for recovering formatting, but it needs to be improved and integrated with the tools to be useful.

Behavioral Reflection

There is scope for improving link composition. The current scheme is very simple and does not provide for any control of how links are composed in the case where multiple links need to be installed at the same node. The order of installation of the links defines the order in which the meta-object is activated. As a first step, we plan to realize the link composition features found in Reflex [124]. These allow both ordering and nesting of links to be defined by the client using the framework.

The selection of nodes where a link is to be installed (spatial selection) is done in the current version by iterating the structure. We think that a declarative form for specifying sets of nodes is needed. In the current

system, the client using GEPPETTO 2 has to deal with newly loaded or changed code. A declarative form of spatial selection will provide a general way to deal with new and changed code. The ongoing work on AOP will be interesting in this context: we plan to explore the work that is currently done to support AOP pointcuts to improve spatial selection.

Meta-context

For now, we have used the concept of *context* just to make the meta computation distinguishable from the base computation. We plan to extend the notion of contextual control of reification to other kinds of contexts then the meta-context. We are working on an implementation of *Context-Oriented Programming* [78] based on the work presented in the dissertation. We have experimented in the past with the idea of a first class model of change for programming languages [41]. We will explore the idea of context for structural reflection to model change.

Virtual machine support for meta-contexts is interesting for two reasons. First, we hope to be able to improve performance by realizing all context setup code in the virtual machine. The current implementation is realized without any change to the virtual machine. As the link-calling code is potentially executed at any operation, it is important to make it as efficient as possible. Code executed by the virtual machine is faster by an order of magnitude. Second, as we explained in Section 6.5, the setup code executed when dealing with contexts has to be managed in a particular way: we provide copies of all that code. We plan to implement all this special code in the virtual machine so that it is not part of the structure of the system that is visible to reflection.

An interesting question is how a context-aware reflective language kernel would look like and what the consequences for the language runtime and especially the reflective model would be. We plan to explore this notion of a context-aware reflective language kernel in the future.

Appendix A

The REFLECTIVITY System

A.1 Introduction

The chapter gives a practical, hands-on introduction to the REFLECTIVITY System. We first show how to obtain a running system. We explain the preferences existing for enabling/disabling and configuring REFLECTIVITY, at the end we show some basic examples of using GEPPETTO 2.

A.2 Installation

A.2.1 Downloading a Pre-built Version

The easiest way to obtain a ready-to-run version is to download the pre-built Squeak image from:

http://www.iam.unibe.ch/~scg/Research/Reflectivity/

Besides the image file, a current VM is needed. This can be obtained from http://www.squeak.org.

A.2.2 Building from Scratch

REFLECTIVITY can be installed in an existing Squeak 3.9 image. To rebuild an image for REFLECTIVITY, the following steps have to be made:

- First install md patches by

  - downloading SqueakPatches-md.zip[1] unpacking and copying the content to your squeak directory.
  - load the patches in Squeak using the filelist browser to navigate to the patches directory, select the file 000-INSTALL.txt. Then execute the contents of the file in the current image. This takes a while.

- In the Monticello Browser, open the SqueakSource repository of REFLECTIVITY:

```
MCHttpRepository
    location: 'http://www.squeaksource.com/Reflectivity'
    user: ''
    password: ''
```

  Then install the ReflectivityLoader from the Monticello Browser and execute:

```
ReflectivityLoader new loadStablePackages
```

  to load the dependencies. Two debugger windows will show up, these are not errors, but the system asking if you want to add variables to base classes. Proceed by pressing *'proceed'* in the debugger window.

- If you want the complete image to use reflective methods, the image needs to be recompiled. This can be done by using the class PERecompiler. Execute the following code from the class comment:

```
[PERecompiler new inspect; recompileImage] forkAt: 30
```

A.2.3 Preferences

A REFLECTIVITY image provides a number of *preferences* to customize the working of the system. These can be set via the *Preference Browser* as shown in Figure A.1.

For Reflectivity, one important category in the preferences browser to look at is named compiler. Here the compileUseNewCompiler preference needs to be enabled, as we build on top of the NewCompiler framework.

The preferences governing REFLECTIVITY are found in the reflectivity category. In addition, the category reflectivitydemo contains preferences related to demos for PERSEPHONE.

The following preferences can be set:

[1] http://www.iam.unibe.ch/~scg/Research/Reflectivity/SqueakPatches-md.zip

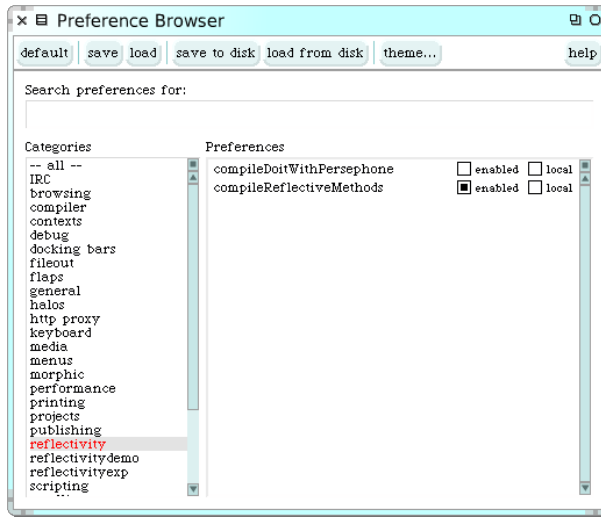

Figure A.1: The preference browser.

compileDoitWithPersephone Evaluate doit-expressions with PERSEPHONE. Experimental.

compileReflectiveMethods Enable PERSEPHONE. All new code is compiled as reflective methods.

A.3 Examples for GEPPETTO

We present some simple examples for the use of GEPPETTO 2. First we show a simple counter and how to count method invocations. Then we look at how to reify message sends and instance variable accesses.

A.3.1 A Simple Counter

The system provides a very trivial counter called GPCounter. It can be instantiated when multiple counters are needed. Alternatively, it provides a simple global counter as a class object:

```
GPCounter reset.
GPCounter inc.
GPCounter count.
```

To use this counter with GEPPETTO, we have to define a link to call the inc method on the counter as a meta-object:

```
link := GPLink new metaObject: GPCounter;
    selector: #inc
```

This link now can be installed in the system. As the link does not request any execution related data, it is compatible with any AST node. We show how it is used on methods, message sends and variable reads.

A.3.2 Method Execution

For example, we can count method invocations. Let's install the counter on a the method + of class Integer:

```
(Integer>>#+) methodNode link: link.
```

The result should be a counter that is counting up slowly while we use our Squeak system. The reason this method is not called very often is that addition of all integers that fit into 31 bits is handled by the class SmallInteger. Addition of big numbers is handled by for example LargePositiveInteger. What we are counting is therefore only the case when a SmallInteger is converted into a large number. Operations on large numbers are slower than SmallInteger calculations and therefore lead to a substantial decrease in performance. The code presented can be used to detect unwanted calculations with large integers.

A.3.3 Message Sends

The GPCounter link can be used to count message sends. We install it on all sends to self in the SystemWindow class:

```
SystemWindow sends do: [:send | send isSelfSend ifTrue:[send link: link]].
```

When inspecting the GPCounter, we can see the counter being incremented when, for example, we click on a window to activate it.

A.3.4 Variables

We can put the link on variables. For example, for counting all read accesses of instance variables. For the following, it is advisable to make sure that the two preferences regarding the meta-context are actually enabled. Code from the class SystemWindow is called quite frequently from the system, and

we can see the problem of *meta-call recursion* in action as soon as we open an inspector on the counter.

```
SystemWindow instanceVariableReads do: [:var | var link: link].
```

After installing the link, we can see the instance variable accesses to objects of class SystemWindow as they happen.

Appendix B

Glossary

Behavioral Reflection. Reflection that is concerned with runtime abstractions, for example message sends or variable accesses, is called *behavioral reflection*.

BYTESURGEON. A runtime bytecode transformation system for Smalltalk. The first version of GEPPETTO uses BYTESURGEON to insert *hooks* into the bytecode of methods. More information in Chapter 3 or the paper on runtime bytecode transformation [40].

Causally connected. A system is said to be *causally connected* to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect of the other and vice versa [97].

GEPPETTO. The realization of Reflex in Squeak Smalltalk is called GEPPETTO. It provides *partial behavioral reflection*. There are two versions: the first was based on bytecode transformation (using BYTESURGEON) and closely follows the original Reflex implementation. The second version leverages *sub-method reflection*. *Links* are annotations on the sub-method structure.

Hook. A short piece of code inserted in the bytecode that realizes the call to the *meta-object*. Hooks are inserted using a bytecode instrumentation, *e.g.*, Javassist [28] for Java or BYTESURGEON [40] for Smalltalk.

Hookset. A collection of *operation occurrences* where *hooks* are installed. A Hookset is bound to a meta-object with a *link*. Hooksets support gathering of heterogeneous execution points, *i.e.*, occurrences of different operations in different classes and methods.

Intercession. The ability of a system to provide a *causally connected* representation of itself that can be *changed* from within the system is called *intercession*. A system providing *introspection* and *intercession* is called *reflective*.

Introspection. The ability of a system to provide a representation of itself that can be *queried* from within the system is called *introspection*. Only when this representation can in addition be changed (see *intercession*) we call a system *reflective*.

Link. In general, the term *link* denotes the connection of the base with the meta-level. It is reified in *partial behavioral reflection* as an object. The link here has multiple functions: it defines the meta-object and the protocol between the base and the meta (which method to call and which information to pass as arguments). In addition, the link is controlled by a condition. See Chapter 3 and Chapter 5.

MetaclassTalk. A CLOS-style meta-class oriented MOP for Smalltalk [17].

Meta-object Protocol. "Metaobject protocols are *interfaces* to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language." [88]

MOP. see *Meta-object Protocol*.

PBR. see *Partial Behavioral Reflection*.

Partial Behavioral Reflection. A fine-grained and efficient MOP. The reifications are precisely selectable in spatial and temporal dimensions. The meta-level behavior is flexibly engineered by means of fine-grained protocols, selection can span different operations over multiple classes (heterogeneous operation occurrences). Provided by GEPPETTO (Smalltalk, see Chapter 3 and 5) or Reflex [128] (for Java).

Reification. The conversion of an interpreter component into an object which the program can manipulate [59] is called *reification*

Reflection. A *reflective system* is a system which incorporates causally connected structures representing (aspects of) itself [97].

Reflex. The implementation of *partial behavioral reflection* for Java [128]. It realizes behavioral reflection with bytecode transformation provided by Javassist [28]. Reflex is the original implementation of partial behavioral reflection.

Structural Reflection. Reflection concerned with the structure of a system is called *structural reflection*. Examples are object-oriented systems where classes and methods are objects.

Sub-method Reflection. Reflection on a sub-method abstraction level, for example instance variable accesses is called *sub-method reflection*. In this thesis the term *sub-method reflection* is often used synonymously with sub-method *structural* reflection, that is the availability of a structural representation of the method bodies.

TREENURSE. An instrumentation framework in the spirit of BYTESURGEON, but works on the AST provided by *sub-method reflection*. The code to be inserted is represented as annotations on the sub-method structure. TREENURSE is a case-study to validate sub-method reflection (see Chapter 4). It has been replaced by a sub-method version of GEPPETTO, see Chapter 5.

TYPEPLUG. A Pluggable type-system [22] for Smalltalk. It uses annotations provided by sub-method reflection and servers as an example (See Section 4.4.2) of how the extended reflective representation is used for language experiments. The master's thesis of Haldimann [69] and a conference paper [67] provide more in-depth information.

Unanticipated Partial Behavioral Reflection. *Partial behavioral reflection* that allows the definition and retraction of reflective features at runtime is called unanticipated partial behavioral reflection. See GEPPETTO.

UPBR. see *Unanticipated Partial Behavioral Reflection*.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT electrical engineering and computer science series. McGraw-Hill, 1991.

[2] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM'05)*, pages 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press.

[3] 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.

[4] Kent Beck. Instance specific behavior: Digitalk implementation and the deep meaning of it all. *Smalltalk Report*, 2(7), May 1993.

[5] John K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 318–330, December 1987.

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

[7] Alexandre Bergel and Stéphane Ducasse. Scoped and dynamic aspects with Classboxes. *Revue des Sciences et Technologies de l'Information (RSTI) — L'Objet (Numéro spécial : Programmation par aspects)*, 11(3):53–68, November 2005.

[8] Alexandre Bergel and Stéphane Ducasse. Supporting unanticipated changes with Traits and Classboxes. In *Net.ObjectDays (NODE'05)*, pages 61–75, Erfurt, Germany, September 2005.

[9] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[10] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.

[11] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

[12] Alexandre Bergel, Stéphane Ducasse, and Lukas Renggli. Seaside – advanced composition and control flow for dynamic web applications. *ERCIM News*, 72, January 2008.

[13] Alexandre Bergel, Robert Hirschfeld, Siobhàn Clarke, and Pascal Costanza. Aspectboxes — controlling the visibility of aspects. In Markus Helfert Joaquim Filipe, Boris Shiskov, editor, *In Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2006)*, pages 29–38, September 2006.

[14] Andrew P. Black and Mark P. Jones. Perspectives on software. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-oriented Systems*, 2000.

[15] Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding infinite recursion with stratified aspects. In Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk, editors, *GI-Edition Lecture Notes in Informatics "NODe 2006 GSEM 2006"*, volume P-88, pages 49–64. Gesellschaft für Informatik, Bonner Köllen Verlag, 2006.

[16] Alan H. Borning and Daniel H.H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings at the National Conference on AI*, pages 234–237, Pittsburgh, PA, 1982.

[17] Noury Bouraqadi. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects (A Smalltalk MOP for the Study of Metaclass Composition and Compatibility. Application to Aspect-Oriented Programming - In French)*. Thèse de doctorat, Université de Nantes, Nantes, France, jul 1999.

[18] Noury Bouraqadi. Concern oriented programming using reflection. In *Workshop on Advanced Separation of Concerns — OOPSLA 2000*, 2000.

[19] Noury Bouraqadi. Safe metaclass composition using mixin-based inheritance. *Journal of Computer Languages, Systems and Structures*, 30(1-2):49–61, April 2004.

[20] Noury Bouraqadi, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In *Proceedings OOPSLA '98*, pages 84–96, 1998.

[21] Noury Bouraqadi, Abdelhak Seriai, and Gabriel Leblanc. Towards unified aspect-oriented programming. In *Proceedings of 13th International Smalltalk Conference (ISC'05)*, 2005.

[22] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[23] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of OOPSLA '04, ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[24] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[25] Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP '89*, pages 109–129, Nottingham, July 1989. Cambridge University Press.

[26] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of Adaptable and Extensible Component Systems*, Grenoble, France, November 2002.

[27] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 285–299, October 1995.

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

[29] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

[30] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *In Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *LNCS*, pages 364–376, 2003.

[31] Elliot Chikofsky and James Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[32] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE Computer Society, 2007.

[33] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, October 2005. ACM.

[34] Pascal Costanza and Robert Hirschfeld. Reflective layer activation in ContextL. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1280–1285, New York, NY, USA, 2007. ACM Press.

[35] M. Dahm. Byte code engineering. In *Proceedings of Java-Informations-Tage (JIT'99)*, pages 267–277, Düsseldorf, Deutschland, sep 1999.

[36] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 219–234. USENIX, 1998.

[37] Steve Dekorte. Io: a small programming language. In Ralph Johnson and Richard P. Gabriel, editors, *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2004, San Diego, CA, USA*, pages 166–167. ACM, 2005.

[38] Marcus Denker. Entwurf von optimierungen für squeak, 2002. Studienarbeit, Universität Karlsruhe.

[39] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):231–251, October 2007.

[40] 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.

[41] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.

[42] Marcus Denker, Orla Greevy, and Michele Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.

[43] Marcus Denker, Orla Greevy, and Oscar Nierstrasz. Supporting feature analysis with runtime annotations. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007)*, pages 29–33. Technische Universiteit Delft, 2007.

[44] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237, 2008. To appear.

[45] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 331–347, New York, NY, USA, 1984. ACM.

[46] Sergey Dimitriev. Language oriented programming: The next programming paradigm. *onBoard Online Magazine*, 1(1), November 2004.

[47] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[48] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[49] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[50] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of 12th International Smalltalk Conference (ISC'04)*, pages 231–257, September 2004.

[51] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[52] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.

[53] Stéphane Ducasse, Roel Wuyts, Alexandre Bergel, and Oscar Nierstrasz. User-changeable visibility: Resolving unanticipated name clashes in traits. In *Proceedings of 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 171–190, New York, NY, USA, October 2007. ACM Press.

[54] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In Ralph Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2004, San Diego, CA, USA*, pages 505–518. ACM, 2005.

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

[56] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[57] 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.

[58] Michael Franz and Thomas Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[59] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 348–355, New York, NY, USA, 1984. ACM.

[60] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

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

[62] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[63] Orla Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, May 2007.

[64] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.

[65] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006.

[66] Thomas Gschwind and Johann Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, page 259, Washington, DC, USA, 2003. IEEE Computer Society.

[67] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 183–204. ACM Digital Library, 2007.

[68] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Journal of Computer Languages, Systems and Structures*, 2008. Accepted for publication, to appear.

[69] Niklaus Haldimann. TypePlug — pluggable type systems for Smalltalk. Master's thesis, University of Bern, April 2007.

[70] Abdelwahab 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.

[71] Abdelwahab Hamou-Lhadj and Timothy 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.

[72] Anthony Hannan. Squeak Closure Compiler. http://minnow.cc.gatech.edu/squeak/ClosureCompiler.

[73] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.

[74] Jurgen Herczeg Heinz-Dieter Bocker. What tracers are made of. In *Proceedings of OOPSLA/ECOOP '90*, pages 89–99, October 1990.

[75] Charlotte Herzeel, Kris Gybels, and Pascal Costanza. A temporal logic language for context awareness in pointcuts. In *Proceeding of the Workshop on Revival of Dynamic Languages*, 2006.

[76] Charlotte Herzeel, Kris Gybels, Pascal Costanza, and Theo D'Hondt. Modularizing crosscuts in an e-commerce application in lisp using halo. In *Proceeding of the International Lisp Conference (ILC) 2007*, 2007.

[77] 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.

[78] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.

[79] Christoph Hofer. Implementing a backward-in-time debugger. Master's thesis, University of Bern, September 2006.

[80] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.

[81] Mamdouh H. Ibrahim and Fred A. Cummins. Ksl: A reflective object-oriented programming language. In *Proceedings of the International Conference on Computer Languages*, pages 186–193. IEEE, October 1988.

[82] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.

[83] Java debug interface (jdi). http://java.sun.com/j2se/1.4.2/docs/-jguide/jpda/jarchitecture.html.

[84] Jython. http://www.jython.org/.

[85] Gregor Kiczales. Towards a new model of abstraction in the engineering of software. In *Proc. of IMSA '92 Workshop on Reflection and Meta-Level Architecture*, 1992.

[86] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[87] Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. Metaobject protocols: Why we want them and what else they can do. In *Object-Oriented Programming: the CLOS Perspective*, pages 101–118. MIT Press, 1993.

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

[89] Rainer Koschke and Jochen Quante. On dynamic feature location. *International Conference on Automated Software Engineering, 2005*, pages 86–95, 2005.

[90] Jay Kothari, Trip Denton, Spiros Mancoridis, and Ali Shokoufandeh. On computing the canonical features of software systems. In *13th IEEE Working Conference on Reverse Engineering (WCRE 2006)*, October 2006.

[91] Wilf R. LaLonde and Mark Van Gulik. Building a backtracking facility in Smalltalk without kernel support. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 105–122, November 1988.

[92] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998.

[93] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. Object flow analysis — taking an object-centric view on dynamic analysis. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL'07)*, pages 121–140, New York, NY, USA, 2007. ACM Digital Library.

[94] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Exposing side effects in execution traces. In Andy Zaidman, Abdelwahab Hamou-Lhadj, and Orla Greevy, editors, *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA'07)*, pages 11–17. Technische Universiteit Delft, 2007.

[95] Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC'07)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.

[96] Pattie Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, January 1987.

[97] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[98] Philippe Marschall. Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern, December 2006.

[99] 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.

[100] Jeff McAffer. Engineering the meta level. In G. Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, San Francisco, USA, April 1996.

[101] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *CACM*, 3(4):184–195, April 1960.

[102] Paul L. McCullough. Transparent forwarding: First steps. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 331–341, December 1987.

[103] Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Softw.*, 8(1):49–57, 1991.

[104] Eliot Miranda. A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk. unpublished, 2002.

[105] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 341–346, November 1986.

[106] Renaud Pawlak. Spoon: annotation-driven program transformation — the aop case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005. ACM Press.

[107] Frédéric Pluquet and Roel Wuyts. Evolution persistence for objects. In *Proceedings of the ERCIM Working Group on Software Evolution (2006)*, 2006.

[108] 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.

[109] Stefan Reichhart. Assessing test quality — TestLint. Master's thesis, University Bern, April 2007.

[110] Lukas Renggli. Magritte — meta-described web application development. Master's thesis, University of Bern, June 2006.

[111] Lukas Renggli and Oscar Nierstrasz. Transactional memory for Smalltalk. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 207–221. ACM Digital Library, 2007.

[112] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02)*, page 34, Los Alamitos CA, October 2002. IEEE Computer Society.

[113] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.

[114] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[115] David Röthlisberger. Geppetto: Enhancing Smalltalk's reflective capabilities with unanticipated reflection. Master's thesis, University of Bern, January 2006.

[116] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.

[117] David Röthlisberger and Oscar Nierstrasz. Combining development environments with reverse engineering. In *Proceedings of FAMOOSr 2007 (Ist International Workshop on FAMIX and Moose in Reengineering)*, 2007.

[118] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 72–81, Los Alamitos CA, 2004. IEEE Computer Society Press.

[119] Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.

[120] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of POPL '84*, pages 23–3, 1984.

[121] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. Croquet, a collaboration system architecture, 2003. in: Proceedings of the First Conference on Creating, Connecting and Collaborating through Computing.

[122] Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.

[123] Patrick Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

[124] Éric Tanter. Aspects of composition in the Reflex AOP kernel. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *LNCS*, pages 98–113, Vienna, Austria, March 2006. Springer.

[125] Éric Tanter. On dynamically-scoped crosscutting mechanisms. *ACM SIGPLAN Notices*, 42(2):27–33, February 2007.

[126] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, Brussels, Belgium, April 2008. ACM Press. To appear.

[127] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *LNCS*, pages 227–242, Vienna, Austria, March 2006.

[128] 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.

[129] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering Java semantics via bytecode manipulation. In *Proceedings of GPCE'02*, volume 2487 of *LNCS*, pages 283–89. Springer-Verlag, 2002.

[130] Daniel Vainsencher and Andrew P. Black. A pattern language for extensible program representation. In *Proceedings of PLoP 2006*, 2006.

[131] Ian Welch and Robert J. Stroud. Kava — using bytecode rewriting to add behavioural reflection to Java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology (COOTS'2001)*, pages 119–130, San Antonio, Texas, USA, February 2001.

[132] Norman Wilde and Michael Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[133] Eric Wong, Swapna Gokhale, and Joseph Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.

[134] Yasuhiko Yokote and Mario Tokoro. Experience and evolution of ConcurrentSmalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 406–415, December 1987.

[135] Andy Zaidman and Serge 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'04)*, pages 329–338, Los Alamitos CA, March 2004. IEEE Computer Society Press.

[136] Pascal Zumkehr. Changeboxes — modeling change as a first-class entity. Master's thesis, University of Bern, February 2007.

Curriculum Vitae

Personal Information

Name:	Marcus Denker
Date of Birth:	September 02, 1975
Place of Birth:	Siegen, Germany
Nationality:	German

Education

2004 - 2008:	Ph.D. in Computer Science in the Software Composition Group, University of Bern, Switzerland Thesis title: *Sub-method Structural and Behavioral Reflection*
1996 - 2004:	University of Karlsruhe, Germany Master's degree in Computer Science, May 2004 Thesis title: *Erweiterung eines statischen Übersetzers zu einem Laufzeitübersetzungssystem*
1995:	Abitur, Siegen, Germany

Complete Curriculum Vitae :
http://www.marcus-denker.de/CV.html