Sub-Method Reflection

Marcus Denker, University of Berne, Switzerland
Stéphane Ducasse, University of Savoie, France
Adrian Lienhard, University of Berne, Switzerland
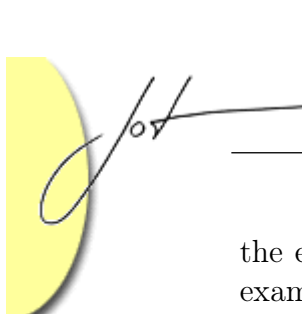**Philippe Marschall**, University of Berne, Switzerland

Reflection has proved to be a powerful feature to support the design of development environments and to extend languages. However, the granularity of structural reflection stops at the method level. This is a problem since without sub-method reflection developers have to duplicate efforts, for example to introduce transparently pluggable type-checkers or fine-grained profilers. In this paper we present PERSEPHONE, an efficient implementation of a sub-method meta-object protocol (MOP) based on AST annotations and dual methods (a compiled method and its meta-object) that reconcile AST expressiveness with bytecode execution. We validate the MOP by presenting TREENURSE, a method instrumentation framework and TYPEPLUG, an optional, pluggable type system which is based on PERSEPHONE.

1   INTRODUCTION

Reflection [37] has proved to be an important property of long-lived and highly-dynamic systems. The literature is full of examples of uses for reflection. For instance, message passing control has been used for a wide range of application analysis approaches, such as tracing [21,9], automatic construction of interaction diagrams, class affinity graphs, test coverage, as well as new debugging approaches [26]. Message passing control has also been used to introduce new language features in several languages, for instance multiple inheritance [6], distribution [2,31], instance-based programming [1], active objects [10], concurrent objects [42], futures [33] and atomic messages [17,29], as well as backtracking facilities [25]. Nowadays, AOP is often using reflection to support its implementation [39,38].

Current object-oriented languages provide access to program structure reifications. In most cases (*e.g.,* Java or C#) only introspection is supported (*i.e.,* these descriptions can be queried). In other languages, intercession is also supported (*i.e.,* the program itself can be changed) [5]. Dynamic languages usually support both introspection and intercession (CLOS [23], Ruby, Python, Smalltalk [17,15]). In such languages, the representation of the program is causally connected to the system itself [27]. When the representation is changed, the running application changes, too and conversely.

The literature also distinguishes structural and behavioral reflection [23]: structural reflection deals with program structure while behavioral reflection deals with

the execution. However, structural reflection often stops at the method level. For example in Smalltalk (the same holds in Java) the only entities supporting limited sub-method structural reflection are collection of bytes (from the compiled method) or characters (from the textual representation of method bodies).

The lack of a higher abstraction of method source code hampers the implementation of tools that require to reflect about it. Examples of such tools are code transformers (*e.g.,* a refactoring tool), pluggable type-checkers, or fine-grained code coverage analysis tools. Since programming languages do not provide support for sub-method reflection each tool has to resort to implement its own infrastructure to represent and reason about source code.

However, this is difficult because (1) text is definitively not structured enough, (2) bytecode is low-level and (3) abstract syntax trees and text are not causally connected to the low-level representation of methods (*i.e.,* changes are not reflected in the bytecode representation of methods).

Another problem is that it is hard to reuse existing infrastructure of one tool in another tool because solutions tend to be very specific. A common infrastructure would not only allow for code reuse but also facilitate data and cache sharing and provide a layer for communication between the tools.

In this paper we propose *reflective methods* which provide a high-level, extensible representation of method bodies accessible through a meta-object protocol (MOP). The high-level representation used by tools is causally connected to its low-level bytecode representation used by the virtual machine (VM). That is, changes to the high-level representation are automatically reflected at the bytecode level. While our implementation is done in Squeak [22], we believe that the concepts can be applied to other languages without severe changes.

The contribution of this paper is: a motivation for sub-method reflection, an analysis of the problems to support it, a MOP for sub-method reflection based on annotated abstract syntax trees and an efficient implementation named PERSE-PHONE. We validate our approach on two non-trivial tools which are implemented based on PERSEPHONE.

This paper is organized as follows. In Section 2 we analyze the difficulties for supporting sub-method reflection. Section 3 presents our solution. After showing case studies (Section 4 and 5) we discuss benchmarks (Section 6) to validate the implementation. After a short overview of related work (Section 7) we conclude in Section 8.

2   CHALLENGES FOR SUPPORTING SUB-METHOD REFLECTION

To support the need 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
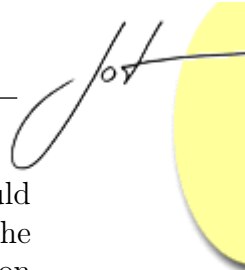
communication between tools. The representation should be persistent: it should not be needed to re-generate it for every tool. We need a high-level model of the method that supports powerful code transformation. Finally, the representation should be causally connected: changing the representation should 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 a sub-method MOP.

Text as Sub-Method Representation

In all current major programming languages, the programmer types text [14, 16]. 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 interface: it lacks the possibility to scope information and manipulate the underlying program elements.

Not causally connected. Changing the method body text 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.

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 properties:

Not persistent. Representations such as ASTs are not persistent: they are 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.

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 [11] and Byte-Surgeon [13])[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 [8]. 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 [8].

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 [41, 32].

Mixed base-level and meta-level code. Code transformation (*e.g.,* bytecode instrumentation [11]) 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.

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

---

[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 to that, ByteSurgeon provides full runtime transformation capabilities: methods can be transformed even in a running system.
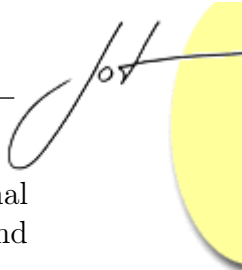
system, (ii) persistent, (iii) extensible and (iv) reasonably compact with minimal performance impact. Furthermore, it should support the separation of base and annotated code and offer a high-level abstraction to the developers.

Such a model for reflective methods would make 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, the AST could be useful. But 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.

3 REFLECTIVE METHODS: ANNOTATED AND CACHED 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*.
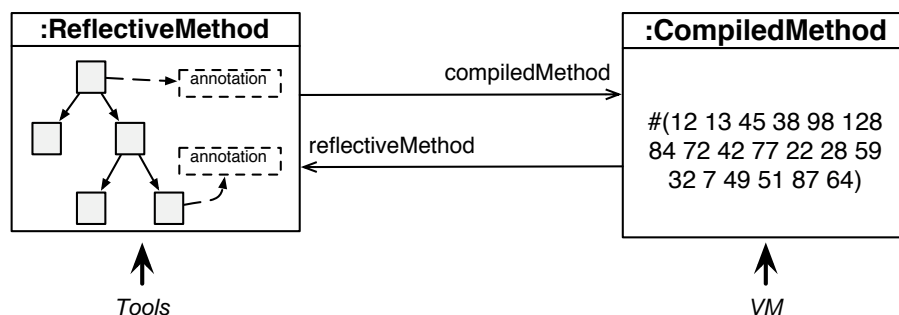

Figure 1: A reflective method is the meta-object of a compiled method.

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 [19]). 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 1 and described in [28].

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 the underworld (the virtual machine), other time as part of the upper world of high-level abstraction.

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.

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 2).
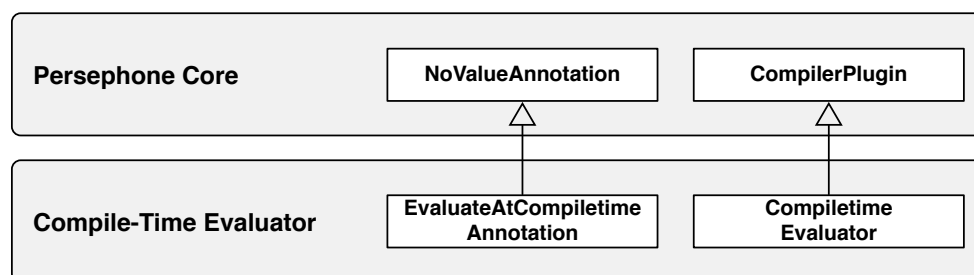

Figure 2: Extension for compile-time evaluation.

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 Compiler-Plugin. 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 annotations 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. It replaces the original node and the visiting process continues.

AST and Tree Transformation API

The AST used in PERSEPHONE is an extended version of the Smalltalk refactoring engine AST [35]. 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/replaced, see Table 1). On top of these simple transformations, the Parse-TreeRewriter 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 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 would have 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, annotations provide a way to define non destructive transformations *i.e.,* we can always identify the original method.

AST Annotations

As discussed in Section 2, 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 [3]. For object state extensions, our MOP provides annotation objects that are attached to any node as shown in Figure 3.
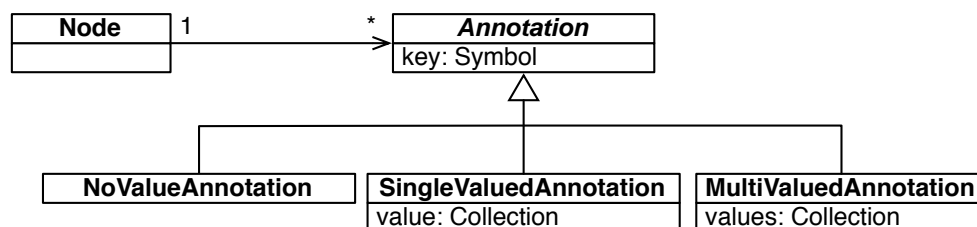

Figure 3: The annotation hierarchy.

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 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.

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
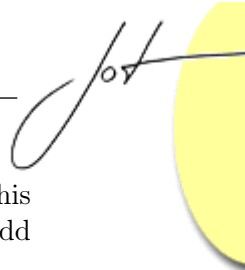
regenerated. For example, when a method is recompiled from source, the nodes of this method will have no annotations, but clients can be notified of any code change and add annotations again if needed.

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 with the instrumentation framework TREENURSE.

Characteristics of the Solution

Now we analyze our approach according to the requirements presented in Section 2.

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 meta-programming.

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 meta-programs 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 6. 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.

We now validate our approach by presenting two tools that we have built based on PERSEPHONE. The first tool is a framework for instrumenting code (TREENURSE). TREE-NURSE is very similar ByteSurgeon [13] or Javassist [11]. Using TREENURSE, we show how the extensibility of our approach is used for communication between tools: we implement a statement coverage tool that records the coverage information as annotations on the reflective method. The second example presents TYPEPLUG, an optional type system that uses textual annotations.

4 VALIDATION: INSTRUMENTATION USING ANNOTATIONS

TREENURSE is a framework for instrumenting code. It is modeled to have an interface very similar to ByteSurgeon, a bytecode transformation framework [13]. In contrast to ByteSurgeon, it works on the AST and is implemented as a PERSEPHONE compiler-plugin.

To present TREENURSE, we start with discussing a simple example. We have a method that does an assignment and we want to annotate the code with trace statements for debugging that increases a counter. The original code and what we want to actually do at runtime are shown in Figure 4.

```
a := 1 max: 3                    a := 1 max: 3.
                                 AssignmentCounter inc.


    Original Code                    Instrumented Code
```

Figure 4: Code instrumented with a counter.

Here it is important to understand that we want to change the semantics of the method, but we don't 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 undistinguishable 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 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
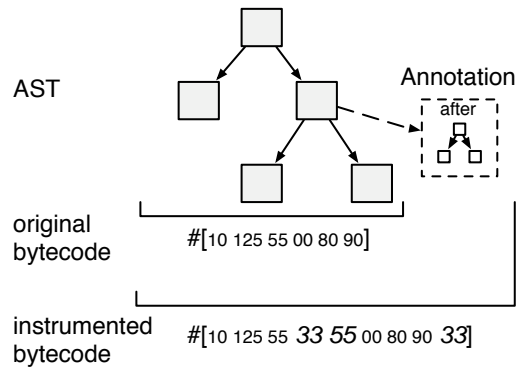
Figure 5: Instrumenting a reflective method with annotations.

block with the instrumentation code. We only 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
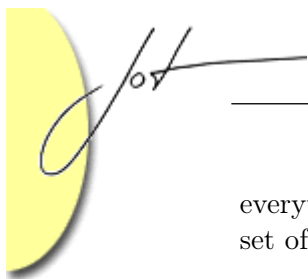
everything else, block arguments have to be used. Each kind of node provides a different set of meta variables that can be used as block arguments:

Node	Metavariable	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 a new value
	variableName	the name of the variable
	value	the value to be assigned to the variable
return node	value	the value 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 a conceptually simple task. Before 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 do using our instrumentation framework:

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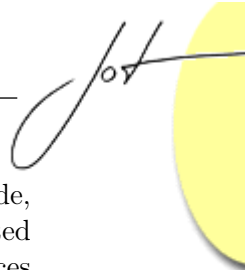
Here we iterate over all the nodes of the method, the block is evaluated for each node, binding the node to the variable eachNode. Before each node, we insert code. The inserted code is described by a block. This block references the meta variable node which references the AST node at runtime.

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.

5 VALIDATION: PLUGGABLE TYPE SYSTEM

TYPEPLUG [20] is an optional, pluggable type system [7] 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 for 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.

6   PERFORMANCE AND MEMORY ANALYSIS

In this section we discuss results of performance benchmarks and memory consumption. The machine used is an Apple MacBook Pro (2Ghz Intel Core Duo, 2GB RAM). We present an analysis of the effect of our caching scheme and discuss memory consumption.

Performance of Cache

To analyze cache performance, we use the *TinyBenchmark* suite that is part of the normal Squeak distribution (Table 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 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 3 shows the memory consumption of the AST of a complete Squeak system.

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

Table 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 then Squeak. Eclipse allocates ca. 180MB of memory when having the ArgoUML source loaded. Thus the memory consumption we see for PERSEPHONE is typical for a modern development environment.

It is also to 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 [18].

7   RELATED WORK

Mirrors

Bracha and Ungar [8] suggest structuring the meta layer with the help of mirrors. In this approach, objects are not reflective, they need a mirror that provides the reflective API. They mention mirrors on methods as future work. Mirror methods and our reflective methods share some similarities, but as mirrors are generated on demand they are not persistent. They provide a high-level view on low-level objects. We instead provide the high level view by default.

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 also support method annotations (also called pragmas).

Spoon [34] 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 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.

---

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

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 [14] argues that programs should no longer be text but a graph described with a metamodel built for a certain kind of problem. The language would be mapped to another one for execution or interpretation. Edwards [16] 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 [4] 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 [30] 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 [12] 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.

Sub-Method Behavioral Reflection and AOP

Behavioral Reflection is concerned with the execution of programs [37]. The elements reasoned about can be those of sub-method abstraction, like message sends and instance variable accesses. Reflection frameworks like Reflex [40] or Geppetto [36] provide a static model for sub-method structure, modeled as a sequence of operations. Thus they provide less complete structural model then that of the AST. For implementation, they rely on bytecode instrumentation.

In AOP [24], a *joinpoint* describes a point in the the execution of a program. Thus AOP deals with an execution model, it does not provide a general, static model of code. The structural model provided by PERSEPHONE in connection with partial behavioral reflection would be an interesting basis for implementing an AOP system.

8   CONCLUSION AND FUTURE WORK

In this paper we have motivated the need for *reflective methods* and presented an implementation called PERSEPHONE. We validated our approach by implementing an instrumentation framework and a pluggable type system.

PERSEPHONE has been used very successfully in a number of projects. The TREE-NURSE instrumentation framework has been used to implement a test coverage tool that provides line-by-line coverage analysis. Additionally, the instrumentation framework is the

basis of an ongoing experiment that analyzes how objects flow through the system. For this, TREENURSE is used to instrument all variable assignments and accesses and message sends.

We plan to explore the use of annotations in the context of behavioral reflection. Geppetto [36] has been implemented using bytecode instrumentation. We are evaluating how Geppetto can use annotations on the AST instead of hooks at the bytecode level.

Compactness of the representation is an interesting field for future work. For our experiments, memory consumption has never been a problem, but nevertheless, we plan to research how to improve storage by optimizing the AST representation leveraging transparent AST compression.

Another interesting direction is to use *reflective methods* to replace text: source code could be reconstructed from the AST. The current version provides some support for recovering formatting, but it needs to be improved and integrated with the tools.

Acknowledgments

REFERENCES

[1] Kent Beck. Instance specific behavior: Digitalk implementation and the deep meaning of it all. *Smalltalk Report*, 2(7), May 1993.

[2] John K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 318–330, December 1987.

[3] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.

[4] Andrew P. Black and Mark P. Jones. Perspectives on software. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-oriented Systems*, 2000.

[5] D.G. Bobrow, R.P. Gabriel, and J.L. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[6] 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.

[7] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] Sergey Dimitriev. Language oriented programming: The next programming paradigm. *onBoard Online Magazine*, 1(1), November 2004.

[15] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[16] 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.

[17] 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.

[18] Michael Franz and Thomas Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[19] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language.* Addison Wesley, 1989.

[20] Niklaus Haldimann. Typeplug — pluggable type systems for smalltalk. Master's thesis, University of Bern, April 2007.

[21] Jurgen Herczeg Heinz-Dieter Bocker. What tracers are made of. In *Proceedings of OOPSLA/ECOOP '90*, pages 89–99, October 1990.
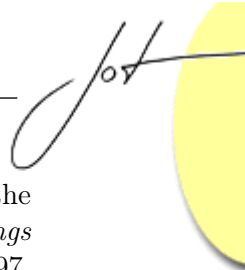
[22] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

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

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[25] 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.

[26] Bill Lewis and Mireille Ducassé. Using events to debug Java programs backwards in time. In *OOPSLA Companion 2003*, pages 96–97, 2003.

[27] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[28] Philippe Marschall. Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern, December 2006.

[29] 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.

[30] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *CACM*, 3(4):184–195, April 1960.

[31] Paul L. McCullough. Transparent forwarding: First steps. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 331–341, December 1987.

[32] Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Softw.*, 8(1):49–57, 1991.

[33] 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.

[34] 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.

[35] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[36] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 47–65. Springer, 2007.

[37] Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.

[38] Éric Tanter. An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.

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

[40] 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.

[41] Daniel Vainsencher and Andrew P. Black. A pattern language for extensible program representation. In *Proceedings of PLoP 2006*, 2006.

[42] Yasuhiko Yokote and Mario Tokoro. Experience and evolution of ConcurrentSmalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 406–415, December 1987.

ABOUT THE AUTHORS

Marcus Denker is a PhD student at the Software Composition Group of the University of Bern. His research focuses on reflection and meta-programming for dynamic languages. Marcus Denker is an active participant in the Squeak open source community for many years. He received a Dipl.-Inform. (MSc) from the University of Karlsruhe/Germany. Contact him at denker@iam.unibe.ch.

Stéphane Ducasse is a Research Director at INRIA Futurs. He was full Professor at the Université de Savoie, where he led the Language and Software Evolution Group. Before that he co-directed with O. Nierstrasz the Software Composition Group of the University of Bern. He is the president of the European Smalltalk User Group and has a lof of fun with Smalltalk. He is involved in the development of Squeak the open-source Smalltalk. His research interests include dynamic languages, reflective systems, reengineering and maintenance, program visualization, and metamodeling. He holds a PhD from the Université de Nice-Sophia Antipolis in computer science. You can contact him at stephane.ducasse@inria.fr.

Adrian Lienhard is a doctoral candidate in computer science at the University of Bern and co-founder of netstyle.ch, a startup company specialized in business Web application development. His research interests include reengineering of object-oriented systems, dynamic analysis, dynamic languages, language design, program visualization and Web applications. He received his MSc in computer science from the University of Bern. Contact him at lienhard@iam.unibe.ch.

Philippe Marschall is Software Engineer at Netcetera, a swiss IT services company. He received his MSc in computer science from the University of Bern. Contact him at philippe.marschall@gmail.com.