Unanticipated Partial Behavioral Reflection

David Röthlisberger[1] Marcus Denker[1] Éric Tanter[2]

[1]Software Composition Group
University of Bern – Switzerland

[2]Center for Web Research/DCC
University of Chile, Santiago – Chile

Abstract. Dynamic, unanticipated adaptation of running systems is of interest in a variety of situations, ranging from functional upgrades to on-the-fly debugging or monitoring of critical applications. In this paper we study a particular form of computational reflection, called *unanticipated partial behavioral reflection*, which is particularly well-suited for unanticipated adaptation of real-world systems. Our proposal combines the dynamicity of unanticipated reflection, *i.e.,* reflection that does not require preparation of the code of any sort, and the selectivity and efficiency of partial behavioral reflection. First, we propose unanticipated partial behavioral reflection which enables the developer to precisely select the required reifications, to flexibly engineer the metalevel and to introduce the meta behavior dynamically. Second, we present a system supporting unanticipated partial behavioral reflection in Squeak Smalltalk, called Geppetto, and illustrate its use with a concrete example of a Seaside web application. Benchmarks validate the applicability of our proposal as an extension to the standard reflective abilities of Smalltalk.

1 Introduction

Dynamic adaptation of a running application makes it possible to apply changes to either the structure or execution of the application, without having to shut it down. This ability is interesting for several kinds of systems, *e.g.,* context-aware applications, long-running systems that cannot afford to be halted, or for monitoring and debugging systems on-the-fly. Adaptation can be considered *a priori* by adopting adequate design patterns such as the strategy pattern [1], but such anticipation is not always possible nor is it desirable: potentially many parts of an application may have to be updated at some point. This is an area in which metaobject protocols, by providing *implicit reification* of some parts of an application [2], are very useful [3–5].

Reflection in programming languages is a paradigm that supports computations about computations, so-called *metacomputations*. Metacomputations and base computations are arranged in two different levels: the *metalevel* and the *base level* [6, 7]. Because these levels are causally connected any modification to the metalevel representation affects any further computations on the base level [8]. In object-oriented reflective systems, the metalevel is formed in terms of metaobjects: a metaobject 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 reifications deal with the *execution* of the program, then we are referring to *behavioral reflection*.

This paper is concerned with a particular form of behavioral reflection, since Smalltalk already supports powerful structural reflective mechanisms. Following the work of McAffer on metalevel engineering [9], we adopt an *operational* decomposition of the metalevel: reifications represent *occurrences* of *operations* denoting the activity of the base program execution. Examples of operations are message sending, method execution, and variable accesses. An occurrence of an operation is a particular event (*e.g.,* a particular sending of a message).

We focus on two particular enhancements of behavioral reflection that make it more appropriate in real-world systems. First, *unanticipated* behavioral reflection (UBR) enables the deployment of metaobjects affecting the behavior of a program while it is already running. This makes it possible to fully support unanticipated software adaptation [4]. Second, an admitted issue of behavioral reflection is its overhead in terms of efficiency: jumping to the metalevel at runtime — reifying current computation and letting a metaobject perform some metalevel behavior — is powerful but costly. *Partial* behavioral reflection (PBR) has been proposed to overcome this issue, by letting users precisely select what needs to be reified, and when [10]. Furthermore, PBR allows for flexible engineering of the metalevel, making it possible to design a *concern-based* metalevel decomposition (*i.e.,* where one metaobject is in charge of one concern in the base application) rather than the typical *entity-based* metalevel decomposition (*e.g.,* one metaobject per object, or one metaobject per class). Hence it is possible to reuse or compose metaobjects of different concerns which greatly eases the engineering of the metalevel [9, 10].

In this paper we propose unanticipated partial behavioral reflection (UPBR) which allows us to insert reflective behavior at runtime into a system (the "unanticipated" in this definition). The reifications are precisely selectable in spatial (which occurrences of which operations) and temporal (when those occurrences are reified) dimensions (the "partial" in UPBR). The metalevel behavior is flexibly engineered by means of fine-grained protocols and selection possibilities that supports gathering of heterogeneous execution points (*i.e.,* occurrences of different operations in different classes and methods).

The contributions of this paper are *(a)* a motivation for the need of unanticipated partial behavioral reflection (UPBR), *(b)* an implementation of UPBR in Squeak Smalltalk, called GEPPETTO, *(c)* an illustration of the use of UPBR in the detection and resolution of a performance bottleneck in an application, without the need to actually stop the application. This is unique because the existing proposals of UBR do not fully support PBR, and reciprocally, the existing systems that truly support PBR are not able to provide full UBR.

The paper is organized as follows: in the next section we describe a running example that serves as the baseline for our motivation and illustration of our proposal. Section 3 then discusses existing reflective support in Smalltalk, as well as the MetaclassTalk extension, followed by an overview of proposals for UBR (Iguana/J) and PBR (Reflex). In Section 4 we describe how we establish an efficient and expressive approach for UPBR in Smalltalk using runtime bytecode manipulation [11]. Section 4.3 is then dedicated to a description of how to use Geppetto, the framework providing UPBR in Smalltalk, by solving our running example. We describe the design of Geppetto in more detail in Section 5. Section 6 discusses some implementation issues and in Section 7 we report on some benchmarks validating the applicability of Geppetto. Section 8 concludes and highlights directions for future work.

2 Running Example

Let us consider a collaborative website (a Wiki), implemented using the Seaside web framework [12]. 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 the 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 natural 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. So to sum up, this example requires *unanticipated partial behavioral reflection* to be solved.

3 Related Work and Motivation

As discussed earlier, changing behavior reflectively at runtime is of great interest for all applications and systems that need to run continuously without interruption, such as servers which provide mission-critical applications. It should be possible to analyze and change the behavior of such a system without the need of stopping and restarting it.

We choose the Smalltalk [13] dialect Squeak [14] 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. We also discuss very related proposals formulated in the Java context, both for unanticipated behavioral reflection and for partial behavioral reflection.

3.1 Reflection in Smalltalk-80

Smalltalk is one of the first object-oriented programming languages providing advanced reflective support [15]. The Smalltalk approach to reflection is based on the metaclass model and is thus inherently structural [7]. A metaclass is a class whose instances are classes, hence a metaclass is the metaobject of a class and describes its structure and behavior. In Smalltalk, message lookup and execution are not defined as part of the metaclass however. Instead they are hard-coded in the virtual machine. It is thus not possible to override in a sub-metaclass 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 presented in [16], or method wrappers [17]. Also, the Smalltalk metamodel 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) [10]. For both reasons (limited expressiveness and lack of partiality), we have to extend the current reflective facilities of Smalltalk: this is precisely the aim of this paper.

3.2 Extended Behavioral Reflection in Smalltalk: MetaclassTalk

MetaclassTalk [18–20] extends the Smalltalk model of metaclasses by actually having metaclasses 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 metaclass 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 [7]: "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 metalevel: 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 metaclass methods if the metaclass indeed provides changed behavior. But even then, once a metaclass defines a custom semantics for an operation, all occurrences of that operation in all instances of the the class are reified. Hence MetaclassTalk provides a less ad-hoc means of doing behavioral reflection than in standard Smalltalk-80, but with a very limited support for partial behavioral reflection.

3.3 Unanticipated Behavioral Reflection: Iguana/J

Iguana/J is a reflective architecture for Java [4] that 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 via a native dynamic library integrated very closely with the Java virtual machine via the Just-In-Time (JIT) compiler interface [4]. 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 metaobject 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 metalevels 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 [4].

3.4 Partial Behavioral Reflection: Reflex

A full-fledged model of partial behavioral reflection was presented in [10]. 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 metalevel is completely configurable: which method to call on the metaobject, 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 (metaobject), also gives total control over the decomposition of the metalevel: a given metaobject can control a few occurrences of an operation in some objects as well as some occurrences of other operations in possibly different objects. Hence metalevel engineering is highly flexible, which makes it possible to directly support a concern-based metalevel decomposition, and this is precisely what is required to support aspect-oriented programming [10, 21].

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 metabehavior. 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.

3.5 Motivation

As we have seen in this section, although unanticipated partial behavioral reflection is highly attractive, no current proposals provide it. Smalltalk-80 is not well-suited for behavioral reflection, MetaclassTalk provides only a limited possibility of metalevel engineering, Iguana/J has limited partiality and implementation limitations, and Reflex has limited dynamicity. Our proposal, a reflective extension of Squeak supporting UPBR called Geppetto, implements the UBR features of Iguana/J and the PBR features of Reflex to form a powerful, open framework for UPBR which extends, enhances and completes the reflective model of Smalltalk in a useful and efficient way.

4 Unanticipated Partial Behavioral Reflection for Smalltalk

We first overview the model of partial behavioral reflection adopted by Geppetto, then discuss how we use bytecode manipulation to achieve unanticipation, and then show how partial behavioral reflection can help to solve the problem introduced in Section 2.

4.1 Partial Behavioral Reflection in a Nutshell

Geppetto adopts the model of partial behavioral reflection (PBR) presented in [10], which we hereby briefly summarize. This model consists of explicit *links* binding *hooksets* to *metaobjects* (Figure 1).

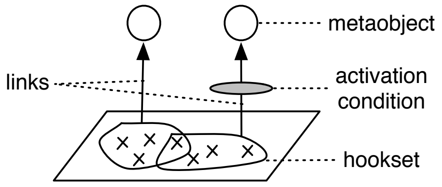

Fig. 1. Links are explicit entities bindings hooksets (at the base level) to metaobjects, possibly subject to activation conditions.

A *hookset* identifies a set of related operation occurrences of interest, at the base level. A *metaobject* 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 metaobject (metalevel). When occurrences of operations are matched by its hookset, the link invokes a method on the associated metaobject, 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 metaobject protocol, and the metaobject can be any object fulfilling this protocol.

Several other attributes further characterize a link, such as the *control* that is given to the metaobject (*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 metalevel 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 2. To profile this application we introduce dynamically a profiler 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 metalevel (*i.e.,* the metaobject, an instance of class Profiler). The link also describes the call to the metaobject, *i.e.,* which method to invoke on the metaobject, specified by passing a metalevel selector.

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

After having installed this link by executing cache install the method #profile: of the metaobject will be executed on every call to method #toughWork of class WikiCore. The developer can provide an arbitrarily complex implementation of the profiler metaobject. See section 4.3 for a more elaborated version of this profiling example.

4.2 Bytecode Manipulation for Unanticipated Behavioral Reflection in Smalltalk

To enable unanticipated partial behavioral reflection in Squeak, the first step is to realize the model for partial reflection as described above. As we have seen in Section 3.1, Smalltalk (and thus Squeak) does not support behavioral reflection properly. To introduce behavioral reflection in a system that does not support it, we can either modify the interpreter (or virtual machine) or transform the code of programs. Modifying the interpreter necessarily sacrifices portability, unless the standard interpreter is actually provided as a sufficiently-*open* implementation.

As Squeak is not implemented using an open interpreter, we use the program transformation approach. We can operate either on source code or on bytecode, but the important thing is, transformation should possibly be done while the program is running. The most appropriate way is arguably to work on bytecode, because it does not require the source code to be present. Squeak by itself does not however support runtime bytecode manipulation appropriately. Fortunately, most of the authors have been involved in ByteSurgeon, a system for runtime bytecode manipulation in Squeak [11].

Following the principles of the implementation of Reflex for Java, we can therefore introduce reflective abilities via insertion of hooks into bytecode. But as opposed to Reflex, in Squeak this can be done at runtime. Since Smalltalk fully supports structural reflection at runtime, and ByteSurgeon extends these structural abilities with method body transformation, we can dynamically introduce selective reflective abilities in running programs.

4.3 Solving the Running Example with Geppetto

To illustrate the use of GEPPETTO, we now explain how to solve the problem introduced in Section 2. In order to find out where the performance issue comes from, we start by elaborating a metaobject protocol to profile the Wiki application. Once we identified the expensive 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: first, the specification of the places where metabehavior is required (*e.g.,* in which classes and methods, for which objects) by configuring a hookset. Second, the definition of the metaobject protocol (*e.g.,* which data is passed to which metaobject) by setting up one or more links. Third and finally, the installation of the defined reflective functionality.

For profiling method execution times of our Wiki application, we need to define a link, binding the appropriate hookset to a Profiler metaobject. 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 metaobject 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, know the metaobject to invoke, an instance of class Profiler:

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

The profiler therefore needs to receive as parameters the selector being called, the currently-executing instance, and the arguments. Its method to call is thus profileMethod:in:withArguments:. This protocol is described by sending the following message to the profile link:

```
profile metalevelSelector: #profileMethod:in:withArguments:
        parameters: {Parameter selector. Parameter self. Parameter arguments.}
        passingMode: PassingMode plain.
```

The class Parameter is used to describe exactly which information should be reified and how it is passed to the meta level. See Section 5 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:

```
profile install.
```

and Geppetto inserts all required hooks. From now on, all method executions in the Wiki application get reified and the Profiler metaobject starts gathering data.

Now suppose that based on the gathered data, we determine that a particular method is indeed taking much time: #toughWork: of our Wiki Worker objects. 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, going back to normal execution, without reification at all. This is achieve by simply executing:

```
profile uninstall.
```

Geppetto then dynamically removes all hooks from the application code, hence further execution is not subject to any extra slowdown at all.

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: Worker; inMethod: #toughWork:; 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 sole piece of information that is reified is the first argument passed to the #toughWork: method, denoted with Parameter arg1.

Cache is a Smalltalk class whose instances manage caching (based on single parameter values). 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 metaobject. Else, the metaobject proceeds with the replaced operation 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: arg
    | result |
    (self cacheContains: arg) ifTrue: [^self cacheAt: arg].
    result := self proceed.
    self cacheAt: arg put: result.
    ^result
```

In order to be able the to proceed with the original operation the class of the metaobject has to inherit from the generic class ProceedMO. Every instance of subclasses of ProceedMO is allowed to proceed with the replaced operations.

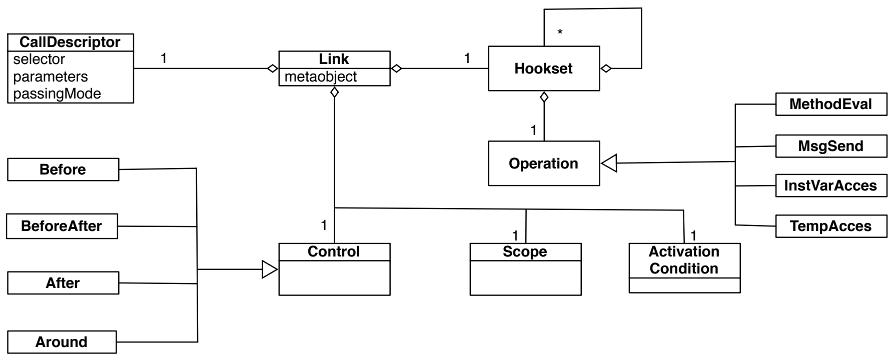

Fig. 2. Class diagram of Geppetto design

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 #toughWork: method are optimized by caching.

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.

5 Geppetto Design

Geppetto instantiates the model of partial behavioral reflection previously presented, as summarized on Figure 2. A link binds a hookset to a metaobject, 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 illustrated on Figure 1. 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*), and/or to a set of methods (using a *method selector*). Convenience methods are provided when an enumerative style of specification is preferred.

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.

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
Operation Occurrence	hookset operationSelector: [:send | send selector = #size]

Table 1. Spatial Selection in GEPPETTO

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 6.3 for details about hook composition.

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

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

The block given for the metaobject creator is evaluated to bootstrap metaobject references. As a shortcut, one can directly give a metaobject 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 metaobject associated to the link is given control over an operation occurrence: it can be either Before, After, BeforeAfter or Around. BeforeAfter means that the metaobject is called *before* and *after* the original operation, whereas Around replaces the operation. The replaced operation then can be executed by calling proceed, if the metaobject is an instance of a subclass of ProceedMO.
– Scope determines the association scheme of a metaobject with respect to base entities. For instance, if the link has object scope, then each instance affected by the link has a dedicated metaobject for the link. The scope can also be *class* (one metaobject per class), or *global* (a unique metaobject 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 metaobject 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.
– a CallDescriptor defines the communication protocol with the metaobject. 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 2 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 2. Supported reified information

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).

6 Implementation Issues

In this section we explain a crucial part of the implementation of Geppetto: the installation of hooks into the bytecode. As explained earlier, we have to dynamically install hooks at runtime to be able to apply reflection in an unanticipated manner into a running system. Therefore, we require a means to manipulate bytecode at runtime. For that purpose we use ByteSurgeon, a framework for runtime manipulation of bytecode in Squeak [11]. 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.

6.1 Adapting Method Binaries

To adapt the binary code of method, 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 [11]).

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 (more in [11]).

6.2 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:

- Checking if the link is active for the currently-executing object;
- Reifying dynamic information and packing this information as specified by the call descriptor of the link;
- Performing the actual delegation to the metaobject, 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 4.3, 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 delegation occurs: the metaobject is retrieved from the link, and the cacheFor: message is sent with first argument as parameter.

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 metaobject requires passing the currently-executing object).

6.3 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 also solving such a conflict automatically at runtime during the installation of every new link.

Detecting a hook conflict is a two-fold process: First, GEPPETTO determines for every link being installed, if another link also manipulates a given method, *i.e.,* if metalevel 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 metalevel 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 [22].

7 Evaluation

We now report on preliminary micro-benchmarks that validate the performance of Geppetto by comparing it with other reflective frameworks and architectures. We measure the slowdown of a fully reified message send over a non-reified message send. In Table 3 we compare the reflective systems Iguana/J [4], and MetaclassTalk [23] to Geppetto. The measurement for Iguana/J was taken from [4]. 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. For a more detailed explanation and the source code of the benchmark, see [24].

We are comparing systems to Geppetto that do not provide partial reflection. As mentioned earlier, the real performance gain of partial reflection comes 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.

System	slowdown factor
Geppetto	10.85
Iguana/J	24
MetaclassTalk	20

Table 3. Slowdowns of different reflective systems for the reification of message sends.

Because Iguana/J is using Java, we cannot do a direct time comparison with Geppetto. So we did such a comparison with MetaclassTalk, since both Geppetto and MetaclassTalk are running in the same environment. We implemented for the operations message sending and instance variable access the same metaobject protocol and the same behavior at the metalevel 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 metalevel 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 4 presents the results of 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 to control which information shall be reified, as described in Section 3.2. Hence GEPPETTO, supporting partial reification of information, will perform even better than the 2-to-3 times speedup against MetaclassTalk in cases where not every information about an operation occurrence is required.

To explain why GEPPETTO is so much faster than MetaclassTalk we have to understand that MetaclassTalk wraps every method (using MethodWrappers [17]) by default to allow message receive 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 massage reception in this case, as we requested only a reification of message sending.

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

Table 4. Speedup of GEPPETTO over MetaclassTalk for reified message send and instance variable read access.

These preliminary benchmarks tend to validate 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.

8 Conclusion and Future Work

In this paper, we have motivated a particular form of computational reflection, called *unanticipated partial behavioral reflection*, which is particularly well-suited for unanticipated adaptation of real-world systems. Our proposal combines the dynamicity of unanticipated reflection, *i.e.,* reflection that does not require preparation of the code of any sort, and the selectivity, efficiency and flexibility of partial behavioral reflection. We have presented a system for unanticipated partial behavioral reflection in Squeak , called GEPPETTO, illustrated its use with a concrete example of a Seaside web application. Preliminary benchmarks validate the applicability of our proposal as an extension to the standard reflective abilities of Smalltalk.

In the future, we plan to work mainly in two directions: the first is to improve GEPPETTO itself, the second consists of using it in a number of projects. As far as improvements to GEPPETTO itself are concerned, we plan to explore advanced scoping for reifications (control-flow based, and more generally, contextual) to give the metaprogrammer even more means to control where and when reification should occur. Another track is to redesign the backend of Geppetto: we decided to use bytecode transformation as we could leverage the fast and easy to use ByteSurgeon framework. But bytecode is a very low-level representation means to trade performance with expressiveness. We plan to extend the Smalltalk structural meta model to provide a high-level model of sub-method structure and explore its use for Geppetto. We are currently working on a number of projects that could benefit from Geppetto. We have experimented with back-in-time debugging [25], but the prototype directly uses ByteSurgeon for now; we plan to explore how Geppetto can be used instead. Another interesting possibility is to use Geppetto as the basis for dynamic analysis [26].

Finally, we plan to explore dynamic aspects for Smalltalk with Geppetto. Because as argued in the body of work on versatile kernels for AOP [21,27], the flexible model of partial behavioral reflection on which both Reflex and Geppetto are based is particularly well-suited to serve as an underlying infrastructure for AOP. This would then allow Geppetto to provide more elaborate AOP features than what the other known dynamic AOP systems for Smalltalk [28,29] do at present.

Acknowledgments. We acknowledge the financial support of the Swiss National Science Foundation for the projects “A Unified Approach to Composition and Extensibility” (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006) and “Analyzing, capturing and taming software change” (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008). É. Tanter is partially financed by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

References

1. Gamma, E., Helm, R., Vlissides, J., Johnson, R.E.: Design patterns: Abstraction and reuse of object-oriented design. In Nierstrasz, O., ed.: Proceedings ECOOP ’93. Volume 707 of LNCS., Kaiserslautern, Germany, Springer-Verlag (1993) 406–431
2. Rao, R.: Implementational reflection in Silica. In America, P., ed.: Proceedings ECOOP ’91. Volume 512 of LNCS., Geneva, Switzerland, Springer-Verlag (1991) 251–267
3. Kiczales, G., Ashley, J., Rodriguez, L., Vahdat, A., Bobrow, D.G.: Metaobject protocols: Why we want them and what else they can do. In: Object-Oriented Programming: the CLOS Perspective. MIT Press (1993) 101–118
4. Redmond, B., Cahill, V.: Supporting unanticipated dynamic adaptation of application behaviour. In: Proceedings of European Conference on Object-Oriented Programming. Volume 2374., Springer-Verlag (2002) 205–230
5. Tarr, P.L., D’Hondt, M., Bergmans, L., Lopes, C.V.: Workshop on aspects and dimensions of concern: Requirements on, and challenge problems for, advanced separation of concerns. In Malenfant, J., Moisan, S., Moreira, A.M.D., eds.: ECOOP 2000 Workshops. Volume 1964 of LNCS., Springer (2000) 203–240

6. Smith, B.C.: Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA (1982)
7. Ferber, J.: Computational reflection in class-based object-oriented languages. In: Proceedings OOPSLA '89, ACM SIGPLAN Notices. Volume 24. (1989) 317–326
8. Maes, P.: Computational Reflection. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium (1987)
9. McAffer, J.: Engineering the meta level. In Kiczales, G., ed.: Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96), San Francisco, USA (1996)
10. Tanter, É., Noyé, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In: Proceedings of OOPSLA '03, ACM SIGPLAN Notices. (2003) 27–46
11. Denker, M., Ducasse, S., Tanter, É.: Runtime bytecode transformation for Smalltalk. Journal of Computer Languages, Systems and Structures **32** (2006) 125–139
12. Ducasse, S., Lienhard, A., Renggli, L.: Seaside — a multiple control flow web application framework. In: Proceedings of ESUG International Smalltalk Conference 2004. (2004) 231–257
13. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass. (1983)
14. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, A practical Smalltalk written in itself. In: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press (1997) 318–326
15. Rivard, F.: Smalltalk : a Reflective Language. In: Proceedings of REFLECTION '96. (1996) 21–38
16. Ducasse, S.: Evaluating message passing control techniques in Smalltalk. Journal of Object-Oriented Programming (JOOP) **12** (1999) 39–44
17. Brant, J., Foote, B., Johnson, R., Roberts, D.: Wrappers to the rescue. In: Proceedings European Conference on Object Oriented Programming (ECOOP 1998). Volume 1445 of LNCS., Springer-Verlag (1998) 396–417
18. Bouraqadi, N.: 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 (1999)
19. Bouraqadi, N.: Safe metaclass composition using mixin-based inheritance. Journal of Computer Languages, Systems and Structures **30** (2004) 49–61
20. Bouraqadi, N., Seriai, A., Leblanc, G.: Towards unified aspect-oriented programming. In: Proceedings of ESUG 2005 (13th International Smalltalk Conference). (2005)
21. Tanter, É., Noyé, J.: 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 (2005)
22. Tanter, É.: Aspects of composition in the reflex aop kernel. In: Proceedings of the 5th International Symposium on Software Composition (SC 2006). LNCS, Vienna, Austria (2006) 99–114
23. Bouraqadi, N.: Concern oriented programming using reflection. In: Workshop on Advanced Separation of Concerns – OOPSLA 2000. (2000)
24. Röthlisberger, D.: Geppetto: Enhancing Smalltalk's reflective capabilities with unanticipated reflection. Master's thesis, University of Bern (2006)

25. Lewis, B.: Debugging backwards in time. In: Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003). (2003)
26. Denker, M., Greevy, O., Lanza, M.: Higher abstractions for dynamic analysis. In: 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006). (2006) 32–38
27. Tanter, É., Noyé, J.: Motivation and requirements for a versatile AOP kernel. In: 1st European Interactive Workshop on Aspects in Software (EIWAS 2004), Berlin, Germany (2004)
28. Bergel, A.: FacetS: First class entities for an open dynamic AOP language. In: Proceedings of the Open and Dynamic Aspect Languages Workshop. (2006)
29. Hirschfeld, R.: AspectS – aspect-oriented programming with Squeak. In Aksit, M., Mezini, M., Unland, R., eds.: Objects, Components, Architectures, Services, and Applications for a Networked World. Number 2591 in LNCS, Springer (2003) 216–232