Unanticipated Partial Behavioral Reflection: Adapting Applications at Runtime

David Roethlisberger, Marcus Denker, Éric Tanter

Unanticipated Partial Behavioral Reflection: Adapting Applications at Runtime ⋆

David Röthlisberger [a] Marcus Denker [a] Éric Tanter [b]

[a] *Software Composition Group*
IAM — Universität Bern, Switzerland
[b] *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 web application. Benchmarks validate the applicability of our proposal as an extension to the standard reflective abilities of Smalltalk.

⋆ This is an extended version of a paper published in the Proceedings of the International Smalltalk Conference (ISC) 2006 [1].

We acknowledge the financial support of the Swiss National Science Foundation for the project "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.

Email addresses: roethlis@iam.unibe.ch (David Röthlisberger), denker@iam.unibe.ch (Marcus Denker), etanter@dcc.uchile.cl (Éric Tanter).

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 [2], 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 [3], are very useful [4–6].

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* [7, 8]. Because these levels are causally connected any modification to the metalevel representation affects any further computations on the base level [9]. 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 on the other hand reifications deal with the *execution* of the program, then we refer 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 [10], 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 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 [5]. Second, a recognized 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 [11]. 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 [10, 11].

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, *i.e.,* which occurrences of which operations, and temporal, *i.e.,* 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 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 [12]. Section 5 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 6. Section 7 discusses some implementation issues and in Section 8 we report on some benchmarks we used to validate the applicability of GEPPETTO. After highlighting some directions of future work (Section 9) we conclude with Section 10.

This work is based on a paper published in the Proceedings of the International Smalltalk Conference (ISC) 2006 [1]. We report more thoroughly about the design of GEPPETTO in section 6 and have extended the implementation section. An example has been added to section 5. The description of the other examples has been extended. We strengthen the evaluation (Section 8) by providing more benchmarks and give a more thorough treatment of future work in Section 9.

2 Running Example

Let us consider a collaborative website (a Wiki), implemented using the Seaside web framework [13, 14]. 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. 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 [15] dialect Squeak [16] to implement a dynamic approach to reflection which supports *unanticipated partial behavioral reflection* (UPBR), because Squeak represents a powerful and extensible environment, well-suited to implement and explore the possibilities of UPBR. Before presenting our proposal, we discuss the current situation of reflective support in standard Smalltalk-80 as well as in the MetaclassTalk extension [17–19] . 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 [20]. The Smalltalk approach to reflection is based on the metaclass model and is thus inherently structural [8]. 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 submetaclass 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 [21], or method wrappers [22]. 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) [11]. 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 [17–19] 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 [8]: "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 custom semantics for an operation, all occurrences of that operation are reified in all instances of the class. Hence MetaclassTalk provides a less ad-hoc means of doing behavioral reflection than standard Smalltalk-80, but with a very limited support for partial behavioral reflection.

3.3 Unanticipated Behavioral Reflection: Iguana/J

Iguana/J is a reflective architecture for Java [5] which supports unanticipated behavioral reflection and a limited form of partial behavioral reflection.

With respect to unanticipated adaptation, with Iguana/J it is possible to adapt Java applications at runtime without being forced to shut them down and without having to prepare them before their startup for the use of reflection. However to bring unanticipated adaptation to Java, Iguana/J is implemented via a native dynamic library integrated very closely with the Java virtual machine via the Just-In-Time (JIT) compiler interface [5]. 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 metalevel 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 [5].

3.4 Partial Behavioral Reflection: Reflex

A full-fledged model of partial behavioral reflection was presented in [11]. 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 [11, 23].

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 and discuss second how we use bytecode manipulation to achieve unanticipation.

4.1 Partial Behavioral Reflection in a Nutshell

Geppetto adopts the model of partial behavioral reflection (PBR) presented in [11], which we hereby briefly summarize. This model consists of explicit *links* binding *hooksets* to *metaobjects* (Figure 1).

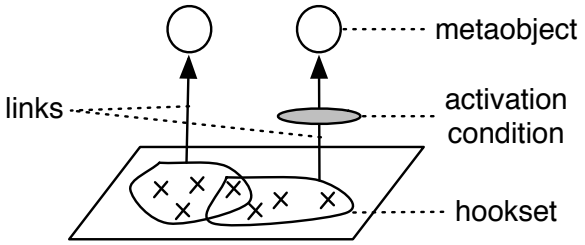

Fig. 1. Links are explicit entities bindings hooksets (at the base level) to metaobjects, possibly subject to activation conditions.

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 dynamically introduce a profiler with GEPPETTO, analyzing the method #toughWork which we suspect of being responsible for the performance issues.

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

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

Second, we specify the link which bridges the gap between the base level (*i.e.,* method #toughWork) and the 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.

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

After having installed this link by executing profiler install the method #profile: of the 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 5 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 point is that 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 however does not support runtime bytecode manipulation appropriately. Fortunately, all of the authors have been involved in ByteSurgeon, a system for runtime bytecode manipulation in Squeak [12].

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.

5 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 costly methods that can be cached, we introduce a caching mechanism with Geppetto.

5.1 Profiling MOP

Defining and introducing dynamically reflective behavior into an application consists of three steps: The first step is the specification of the places where metabehavior is required (*e.g.,* in which classes and methods, for which ob-

jects) by configuring a hookset. In the second step the definition of the metaobject protocol (*e.g.,* which data is passed to which metaobject) is specified by setting up one or more links. Third and finally, we perform the installation of the defined reflective functionality.

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

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

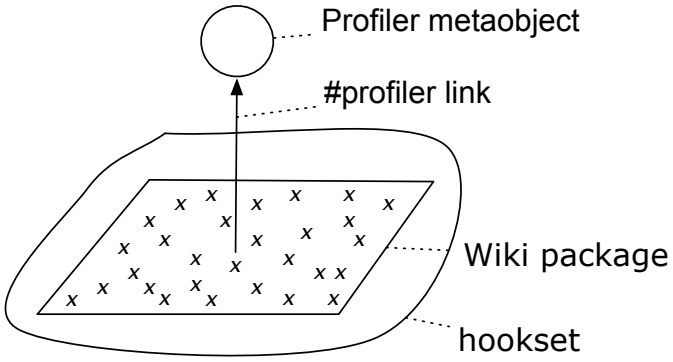

Fig. 2. The profiler hookset affects the whole Wiki application.

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

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

The class Parameter is used to describe exactly which information should be

reified and how to pass it to the metalevel. See Section 6 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.

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

```
"editing a page"
page := WikiModel at: pageName.
page title: newTitle.
doc := DocumentParser parse: wikiText.
page document: doc.
```

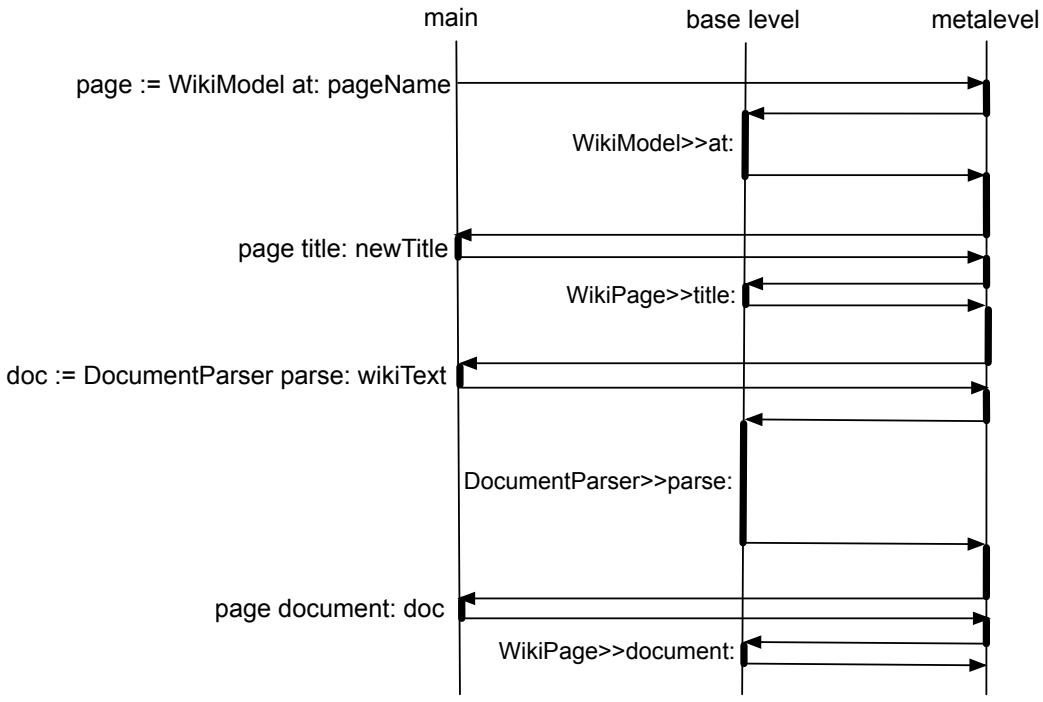

Fig. 3. Execution flow in the Wiki during the editing of a page when the profiler is installed

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

```
profile uninstall.
```

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

5.2 Caching MOP

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

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

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

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

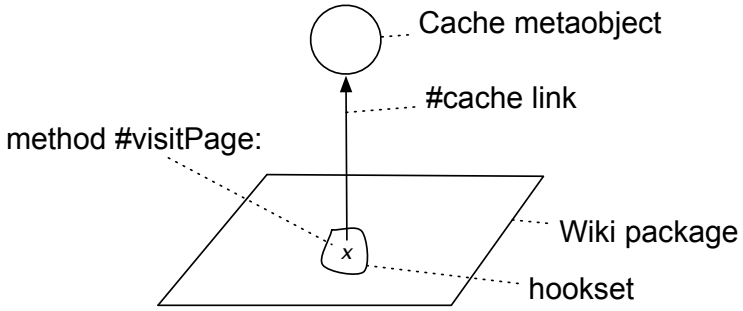

Fig. 4. The cache hookset only affects method #visitPage:.

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

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

Table 1
Effect of the caching meta behavior for rendering one thousand times the HTML code of three wiki pages.

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

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

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

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

The effect of this cache is tremendous. We compare the situation with and without this cache installed by generating one thousand times the HTML code of three exemplary wiki pages with complex content such as links and tables. [1] Without the active caching mechanism the HTML code is completely built up on every single visit to a wiki page, whereas otherwise the HTML code is taken from the cache. As Table 1 shows we achieve an average optimization of almost factor 140 with an installed cache for the result of method #visitPage:.

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

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

5.3 Persistence MOP

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

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

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

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

Next we define a link taking the above hookset and specifying the metaobject and the invocation of it:

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

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

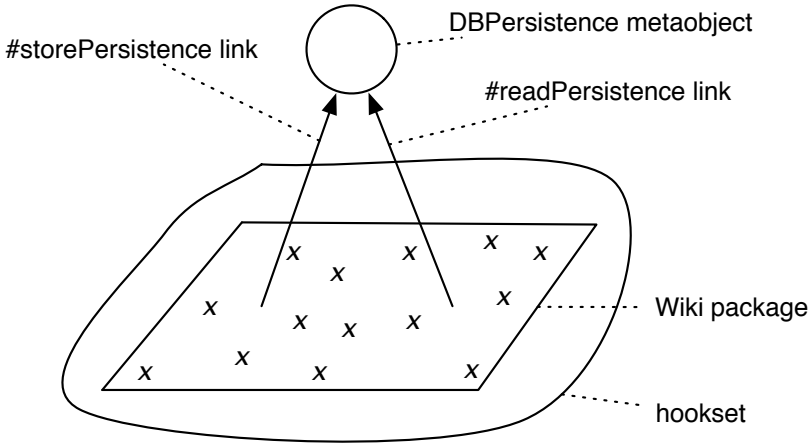

Fig. 5. The persistence hookset encloses the whole Wiki application, but only affects methods containing instance variable accesses (read or write).

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

First, we give the hookset definition:

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

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

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

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

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

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

6 Geppetto Design

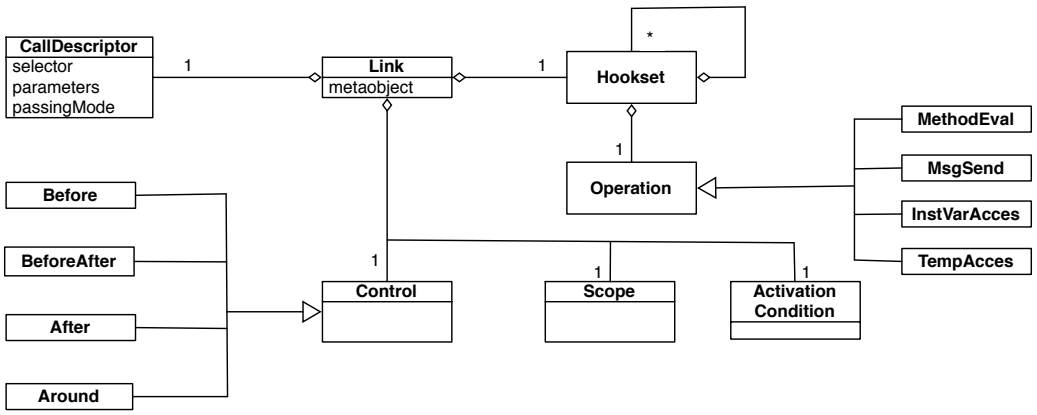

Fig. 6. Class diagram of GEPPETTO design

GEPPETTO instantiates the model of partial behavioral reflection previously presented, as summarized on Figure 6. 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 shown in Table 2. Eventually, occurrences are selected within method bodies (or boundaries), by applying an *operation selector*, *i.e.,* a predicate that can programmatically determine whether a particular occurrence is of interest or not. Coarser levels of selection are provided to speedup the selection process. First of all, one can eagerly specify the operation of which occurrences may be of interest. Furthermore, one can restrict a hookset to a given package, to a set of classes (using a *class selector*), or to a set of methods (using a *method selector*). Convenience methods are provided when an enumerative style of specification is preferred.

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

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

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

Table 2
Spatial Selection in Geppetto

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

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

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

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

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

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

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

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

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

If some hooks of different hooksets conflict with each other, *e.g.,* more than one

hookset affects a particular occurrence of a message send in a given method, then these hooks are automatically composed by GEPPETTO. In a composed hook every single hook is executed in sequence in the order of their installation time. See section 7.3 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 in the metaobject, if this 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 that 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. The predicate defining the activation condition receives the current object (*i.e.,* the object in which the hook is executed) as its sole parameter.
- a CallDescriptor defines the communication protocol with the 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 3 lists all possible parameters depending on the reified operation.

Metaobjects have to be set differently depending on the *scope* attribute of the link. The convenient methods mentioned above, #metaobject: and #metaobjectCreator:, are valid for global scope where the whole link has either one metaobject, or every reflective object has its own metaobject instance. But one can also precisely define which reflective object should have which metaob-

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

Table 3
Supported reified information

ject when using object scope. The method #setMetaobject:forObject: lets us specify which metaobject is valid for which reflective object. Similarly one can use method #setMetaobject:forClass: to associate dedicated metaobjects with reflective classes when using class scope.

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

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

With the following code we inform the link to use this activation condition:

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

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

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

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

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

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

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

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

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

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

7 Implementation Issues

In this section we explain a crucial part of the implementation of Geppetto: the installation of hooks in the bytecode. As explained earlier, we have to dynamically install hooks at runtime to be able to apply reflection in an unanticipated manner to a running system. Therefore, we require a means to manipulate bytecode at runtime. For that purpose we use ByteSurgeon, a framework for runtime manipulation of bytecode in Squeak [12]. 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.

7.1 Adapting Method Binaries

To adapt the binary code of methods, we first select the method in which we want to change the bytecode (recall that a method is defined as the combination of a class and a selector, *e.g.,* WikiPage>>#document). Second, we instrument this method with one of the instrumentation methods added by ByteSurgeon to compiled methods, *e.g.,* #instrumentSends: or #instrumentInstVars:, to access all the specific operations in a method, *i.e.,* message sends or instance variables accesses, respectively. These instrumentation methods expect a block as single argument. In this block we have access to a block argument which denotes the current operation occurrence object. For a message send we get access to an instance of IRSend (this is part of the intermediate representation on which ByteSurgeon is based [12]).

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 [12]).

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

- It checks if the link is active for the currently-executing object;
- It reifies dynamic information and packing this information as specified by the call descriptor of the link;
- It performs 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 5.2, the generated Smalltalk code is:

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

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

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).

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

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

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

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

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

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

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

The proceed statement provided by ByteSurgeon is also used for the resumable metaobjects presented in Section 5.2. To be able to proceed with the original operation in the metaobject Geppetto passes the value of the proceed statement (*e.g.,* a message send or an instance variable access) to the metaobject. This proceed value is stored in the instance variable proceed of the metaobject. By sending the message #proceed to a resumable metaobject, a subclass of ProceedMO, this proceed value is evaluated and the execution of the original operation is triggered, *i.e.,* proceeded. Note that only metaobjects that act around a base level operation can proceed with the original operation.

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

Detecting a hook conflict is a two-step process: First, Geppetto determines for every link that is being installed, if another link also manipulates a given method, *i.e.,* if 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 [24].

System	slowdown factor
Geppetto	10.85
Iguana/J	24
MetaclassTalk	20

Table 4
Slowdowns of different reflective systems for the reification of message sends.

8 Evaluation

We now report on preliminary micro-benchmarks that validate the performance of Geppetto by comparing it with other reflective frameworks and architectures. Subsequently we conduct a more complex benchmark measuring the efficiency of the profiler we presented in Section 5.1 by comparing the execution of some test suites of the Wiki application with and without the profiler being installed in the Wiki.

8.1 Micro-Benchmarks

For the first micro-benchmark we measure the slowdown of a fully reified message send over a non-reified message send. In Table 4 we compare the reflective systems Iguana/J [5], and MetaclassTalk [25] to Geppetto. The measurement for Iguana/J was taken from [5]. 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 [26].

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

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

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

Table 5
Speedup of Geppetto over MetaclassTalk for reified message send and instance variable read access.

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 5 presents the results of the benchmark. The Windows PC mentioned above was also used to execute this benchmark. For both operations, message send and instance variable access, we reified almost every possible information in Geppetto to get a reliable comparison with MetaclassTalk which does not support controlling which information should be reified, as described in Section 3.2. Hence Geppetto will perform even better than the 2-to-3 times speedup compared with MetaclassTalk in cases where not all information about an operation occurrence is required.

The reason why Geppetto is so much faster than MetaclassTalk lies in the underlying mechanisms. MetaclassTalk wraps every method (using MethodWrappers [22]) by default to allow all message receives to be reified even when called from a class not under the control of MetaclassTalk. Geppetto on the other hand does not try to provide reified massage reception in this case, as we requested only a reification of message sending.

8.2 Benchmarking the Profiler

We conducted a third benchmark denoting the general slowdown caused by reflective behavior introduced with Geppetto. We go back to the Wiki example of Section 5.1 where we installed a profiler in this application and compare now the execution times of the Wiki test suite with and without the profiler being active. The test suite contains 131 unit tests. The profiler itself acts around every method evaluation in the Wiki parts Structure as well as Visitor and measures the time required to execute the methods in these packages by stopping the time before and after the execution of the methods. The execution of the original method is triggered in the profiler metaobject by using the proceed statement explained in Section 5.2. The obtained execution time is then stored in a dictionary with the profiled methods as a key and a collection of measured times as value. As denoted in Section 5.1 we only reify the selector, the arguments and the class to which the method belongs and pass

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

Table 6
Overall slowdown caused by the profiler in the metalevel

this information in plain mode to the profiler.

Table 6 contains the results obtained by running the benchmark on the Wiki server also used in Section 5.2 to measure the efficiency of the caching meta behavior. Clearly, the active profiler causes a slowdown between factor 3 and 8. Further benchmarks show that more than 50% of this slowdown is caused by the execution of the profiling code itself, which means that the reification and the invocation of the metaobject is not as much responsible for the high costs of this metalevel profiler as the profiler implementation itself.

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.

9 Future Work

In the future, we plan to focus mainly in two directions: first, we plan to improve Geppetto itself, the second is to explore its usefulness by integrating it in a variety of applications. 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 [27–29]. Another track is to redesign the underlying mechanisms of Geppetto to install reflective behavior: we decided to use bytecode transformation as we could leverage the fast and easy-to-use ByteSurgeon framework. However 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 [30], but our prototype uses ByteSurgeon directly [31, 32]; we plan to explore how Geppetto can be used instead.

Another interesting application is to use Geppetto as the basis for dynamic analysis [33]. We propose in [34] to use Geppetto as an abstraction layer

for dynamic analysis tools. The positive effect is twofold: on the one hand it provides us with a standard API for all dynamic analysis based tools to use, on the other hand it allows the tool developer to abstract from the actual implementation technique.

Furthermore, we want to extend Geppetto so that it not only supports behavioral but also structural reflection by extending it with interfaces to the standard model for structural reflection in Smalltalk. Having the means in Geppetto to perform structural analysis or modifications on running applications completes the reflective support of our framework and allows the user to do both structural and behavioral reflection with the same interface in the same framework.

We consider mechanisms that are currently available to select which operations to be reified as inadequate. We think we can improve current APIs and in addition to that we plan to provide a language to select operations, related to the current work done in Reflex [35] and other systems like JTL [36] or InjectJ [37]. A related project for Geppetto is to study the usefulness of a graphical interface to insert hooks into a running application by selecting operations and operation occurrences interactively. The same graphical environment should also be capable of presenting information about currently installed reflective behavior in base level entities to gain a certain degree of overview about the current reflective status the application and the system itself is in.

Finally, we plan to explore dynamic aspects for Smalltalk with Geppetto. Because as argued in the body of the work on versatile kernels for AOP [23,38], 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 [39, 40] do at present.

10 Conclusion

In this paper, we have motivated a particular form of computational reflection, called *unanticipated partial behavioral reflection*, which is particularly well-suited to 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 pointed out how well the advantages of partial behavioral reflection, *i.e.,* the precise selection of required reifications or the flexible means to engineer the metalevel, can be combined with unanticipated reflection where the meta behavior is introduced dynam-

ically. We have presented a system for unanticipated partial behavioral reflection in Squeak, called Geppetto. We illustrated its use with a concrete example of a Seaside web application which profited in several areas of the use of unanticipated partial behavioral reflection. Preliminary benchmarks validate the applicability of our proposal as an extension to the standard reflective abilities of Smalltalk. Intended future work on the proposed model and system will probably direct our work inter alia to a promising dynamic AOP system in Smalltalk.

Acknowledgments. We thank Stephane Ducasse, Oscar Nierstrasz, Orla Greevy, Lukas Rengli and the anonymous reviewers for their comments.

References

[1] Röthlisberger, D., Denker, M., Tanter, É.: Unanticipated partial behavioral reflection. In: Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006). Volume 4406 of LNCS., Springer (2007) 47–65

[2] 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

[3] Rao, R.: Implementational reflection in Silica. In America, P., ed.: Proceedings ECOOP '91. Volume 512 of LNCS., Geneva, Switzerland, Springer-Verlag (1991) 251–267

[4] 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

[5] 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

[6] 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

[7] Smith, B.C.: Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA (1982)

[8] Ferber, J.: Computational reflection in class-based object-oriented languages. In: Proceedings OOPSLA '89, ACM SIGPLAN Notices. Volume 24. (1989) 317–326

[9] Maes, P.: Computational Reflection. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium (1987)

[10] 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)

[11] 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

[12] Denker, M., Ducasse, S., Tanter, É.: Runtime bytecode transformation for Smalltalk. Journal of Computer Languages, Systems and Structures **32** (2006) 125–139

[13] Ducasse, S., Lienhard, A., Renggli, L.: Seaside — a multiple control flow web application framework. In: Proceedings of 12th International Smalltalk Conference (ISC'04). (2004) 231–257

[14] Renggli, L.: Magritte — meta-described web application development. Master's thesis, University of Bern (2006)

[15] Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass. (1983)

[16] 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

[17] 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)

[18] Bouraqadi, N.: Safe metaclass composition using mixin-based inheritance. Journal of Computer Languages, Systems and Structures **30** (2004) 49–61

[19] Bouraqadi, N., Seriai, A., Leblanc, G.: Towards unified aspect-oriented programming. In: Proceedings of 13th International Smalltalk Conference (ISC'05). (2005)

[20] Rivard, F.: Smalltalk: a reflective language. In: Proceedings of REFLECTION '96. (1996) 21–38

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

[22] 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

[23] 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)

[24] Tanter, É.: Aspects of composition in the Reflex AOP kernel. In Löwe, W., Südholt, M., eds.: Proceedings of the 5th International Symposium on Software Composition (SC 2006). LNCS 4089, Vienna, Austria, Springer (2006) 98–113

[25] Bouraqadi, N.: Concern oriented programming using reflection. In: Workshop on Advanced Separation of Concerns — OOPSLA 2000. (2000)

[26] Röthlisberger, D.: Geppetto: Enhancing Smalltalk's reflective capabilities with unanticipated reflection. Master's thesis, University of Bern (2006)

[27] Nierstrasz, O., Denker, M., Gîrba, T., Lienhard, A.: Analyzing, capturing and taming software change. In: Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06). (2006)

[28] Nierstrasz, O., Bergel, A., Denker, M., Ducasse, S., Gaelli, M., Wuyts, R.: On the revival of dynamic languages. In Gschwind, T., Aßmann, U., eds.: Proceedings of Software Composition 2005. Volume 3628., LNCS 3628 (2005) 1–13 Invited paper.

[29] Tanter, É.: On dynamically-scoped crosscutting mechanisms. In Kniesel, G., ed.: Proceedings of the European Workshop on Aspects in Software (EWAS 2006), Twente, The Netherlands, Technical Report IAI-TR-2006-6, University of Bonn, Germany (2006) 18–22

[30] Lewis, B.: Debugging backwards in time. In: Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003). (2003)

[31] Hofer, C., Denker, M., Ducasse, S.: Design and implementation of a backward-in-time debugger. In: Proceedings of NODE'06. Volume P-88 of Lecture Notes in Informatics., Gesellschaft für Informatik (GI) (2006) 17–32

[32] Hofer, C.: Implementing a backward-in-time debugger. Master's thesis, University of Bern (2006)

[33] Ball, T.: The concept of dynamic analysis. In: Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999). Number 1687 in LNCS, Heidelberg, Springer Verlag (1999) 216–234

[34] 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

[35] Tanter, É.: An extensible kernel language for AOP. In: Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages, Bonn, Germany (2006)

[36] Cohen, T., Gil, J.Y., Maman, I.: JTL: the Java tools language. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications, New York, NY, USA, ACM Press (2006) 89–108

[37] Genßler, T., Kuttruff, V.: Source-to-source transformation in the large. In: Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003. Volume 2789 of Lecture Notes in Computer Science., Springer (2003) 254–265

[38] 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)

[39] Bergel, A.: FacetS: First class entities for an open dynamic AOP language. In: Proceedings of the Open and Dynamic Aspect Languages Workshop. (2006)

[40] 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