Runtime Bytecode Transformation for Smalltalk ⋆

Marcus Denker [a] Stéphane Ducasse [b] Éric Tanter [c]

[a] *Software Composition Group*
IAM — Universität Bern, Switzerland

[b] *Software Composition Group*
IAM — Universität Bern, Switzerland, and
Language and Software Evolution Group
LISTIC — Université de Savoie, France

[c] *Center for Web Research, DCC*
University of Chile, Santiago, Chile

Abstract

Transforming programs to alter their semantics is of wide interest, for purposes as diverse as off-the-shelf component adaptation, optimization, trace generation, and experimentation with new language features. The current wave of interest in advanced technologies for better separation of concerns, such as aspect-oriented programming, is a solid testimony of this fact. Strangely enough, almost all proposals are formulated in the context of Java, in which tool providers encounter severe restrictions due to the rigidity of the environment. This paper present ByteSurgeon, a library to transform binary code in Smalltalk. ByteSurgeon takes full advantage of the flexibility of the Squeak environment to enable bytecode transformation at runtime, thereby allowing dynamic, on-the-fly modification of applications. ByteSurgeon operates on bytecode in order to cope with situations where the source code is not available, while providing appropriate high-level abstractions so that users do not need to program at the bytecode level. We illustrate the use of ByteSurgeon via the implementation of method wrappers and a simple MOP, and report on its efficiency.

Key words: Smalltalk, object-oriented programming, bytecode transformation, metaprogramming

Preprint submitted to Elsevier Science *9 August 2005*

1 Introduction

Many objectives of software engineering can be served by appropriate program transformation techniques. Software adaptation can be used for Binary Component Adaptation (BCA), a technique proposed by Keller and Hölzle which relies on coarse-grained alterations of component binaries to make them interoperable [1]. Another objective of software adaptation is that of separation of concerns [2], as first emphasized by work carried out in the reflection community [3–5], and more recently, aspect-oriented programming (AOP) [6]. In this context, transformation techniques are used to merge together different pieces of software encapsulating different concerns of the global system. Program transformation is a valid implementation techniques for reflection and AOP when an open interpreter of the considered language is not available.

Fine-grained control of computation, such as message passing control in the context of object-oriented programming, is the corner stone of many interesting applications [7]. It has been used for a wide range of application analysis approaches, such as tracing [8–10], automatic construction of interaction diagrams, class affinity graphs, test coverage, as well as new debugging approaches [11,12]. Message passing control has also been used to introduce new language features in several languages, for instance multiple inheritance [13], distribution [14–16], instance-based programming [17], active objects [18], concurrent objects [19], futures [20] and atomic messages [21,22], as well as backtracking facilities [23].

CLOS is one the few languages that offers a dedicated metaobject protocol supporting language semantics customization [24]. Other languages such as Smalltalk and Java rely on techniques or libraries to either transform code or take control of the program execution [7,25]. The most basic way to alter programs is of course to modify the source code and recompile it. This approach is used by several Java systems, such as OpenJava [26] and the Java Syntactic Extender [27]. However, in many contexts, relying on the availability of source code is limiting since most applications ship in binary form, and in open distributed systems, source code is usually not known in advance. Furthermore, the source language from which the actual binary was obtained is not necessar-

⋆ We acknowledge the financial support of the Swiss National Science Foundation for the project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006) and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006). É. Tanter is financed by the Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

Email addresses: denker@iam.unibe.ch (Marcus Denker), ducasse@iam.unibe.ch (Stéphane Ducasse), etanter@dcc.uchile.cl (Éric Tanter).

ily the mainstream language of the runtime system. Bytecode manipulation, as done in the Java world by tools such as BCEL [28] and Javassist [29] is a particularly pertinent alternative. The challenge is to provide appropriate high-level abstractions to bytecode transformation, in order to shield users from the burden of working at the bytecode level [25].

To the best of our knowledge, there is no single bytecode transformation tool for the Smalltalk/Squeak environment, in the line of what Javassist represents for the Java world. This is all the more surprising that the Squeak environment actually represents an ideal environment for bytecode transformation. In contrast with Java where full bytecode transformation is only possible at load time, and very severely limited at runtime, Squeak enables the full power of bytecode transformation to be used dynamically. The purpose of ByteSurgeon is precisely to leverage the flexibility of the Smalltalk language and the Squeak environment to provide a backend to designers of toolkits for component adaptation, reflective and metaprogramming, and aspect-oriented programming.

The contributions of this paper are:

- a motivation for the need of a dynamic bytecode transformation framework for Smalltalk, working at appropriate levels of abstraction,
- a framework, called ByteSurgeon, that enables *runtime* bytecode transformation via a two level API,
- a simple MOP that can be used to compare bytecode transformation frameworks.

The paper is organized as follows: Section 2 explains the need for bytecode manipulation at appropriate levels of abstraction, by discussing related work. Then we present ByteSurgeon at work in Section 3. Section 4 details some aspects of the architecture. In Section 5, we validate the interest of our framework via the implementation of two language features: method wrappers [10], and a simple runtime metaobject protocol (MOP) making use of runtime manipulation for dynamic (un)installation of hooks; a first set of benchmarks completes the validation of ByteSurgeon. Section 6 discusses future work and concludes.

2 The Need for Bytecode Manipulation

There are many ways to change the semantics of programs, ranging from code preprocessing to modification of the language runtime environment. If the language runtime is not an open implementation offering an adequate metaobject protocol (MOP) [24], then modifying it directly sacrifices portability; since

mainstream Smalltalk virtual machines such as Squeak are not open in this sense, we discard the alternative of intervening at the VM level.

Source code transformation can be done either directly on the text (concrete syntax) or on the abstract syntax tree (abstract syntax). Furthermore, in language environments where source code is compiled to an intermediate bytecode language which is abstract enough, bytecode transformation is an interesting approach; it is actually widely used in the Java community.

In Section 2.1 we discuss the inconveniences of source code approaches. Still, once bytecode transformation is agreed upon, the issue of the abstraction level offered to the programmer appears, discussed in Section 2.2. We also discuss the limitations of bytecode transformation in the context of Java.

2.1 Disadvantages of Source Code Transformation

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

No access to the source code. For the sake of saving space or ensuring a first level of privacy, the source code of an application is usually not distributed. Using source code strippers or removing symbolic information are current practices to reduce the size of an application before deployment. Furthermore, in open contexts such as mobile agent platforms and open distributed systems, code is typically not known in advance. One can of course rebuild an AST from bytecode, but this technique presents a number of challenges: bytecode-to-AST decompiling is a slow process, and typically requires the decompiler to know about bytecode generation patterns used by the compiler so as to rebuild meaningful AST nodes.

No original language warranty. Most mainstream languages today, such as Java, Squeak and C#, are based on a virtual machine executing bytecodes, and these virtual machines are actually used as the execution engines of various languages, other than the "original" ones. For instance, for the Croquet environment [30], a number of experimental scripting languages have been developed, among them languages similar to JavaScript and LOGO. Another example is the Python language, which can be compiled to Java bytecodes [31]. To provide practical performance, these languages come with their own custom compiler that produces bytecode for a production-quality virtual machine. Therefore a code transformation tool working at the AST level rebuilding the AST from bytecode would require a custom decompiler. On the other hand, working on bytecode, although lower-level than AST, makes it possible to uni-

formly apply transformations even in the presence of non-original languages.

Recompiling is slow. Finally, transforming source code means that a compiling phase is necessary afterwards to regenerate bytecodes. Recompilation is a slow process, much slower than manipulating bytecode; benchmarks of Section 5.3 validate this statement.

2.2 Bytecode Transformation Approaches

Due to the many reasons explained above, a wide variety of tools have been proposed that rely on bytecode transformation. Surprisingly, most of these tools have been made for Java, and we are aware of very few related proposals in the Smalltalk world.

Java and Bytecode Transformation. The Java standard environment only allows for bytecode transformation at load time. At runtime, it is only possible to dynamically generate new classes from scratch, not to modify existing ones. These restrictions have been somehow relaxed in the context of the JVM debugger interface (JDI) [32], but relying on the debugger interface is not reasonable in a production environment. Furthermore, the possibilities of class reloading are strongly limited as, for instance, new members cannot be added to classes. Using load-time transformation in Java also raises a number of subtle issues related to class loaders and the way they define namespaces in Java [33].

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

Tools like BCEL [28] and ASM [34] strictly reify bytecode instructions: as a consequence, users have to know the Java bytecode language very well and have to deal with low-level details such as jumps and alternate bytecode instructions (a Java method invocation can be implemented by several bytecode instructions, depending on whether the invoked method is from an interface, is private, etc.).

On the contrary, Javassist [35] and Jinline [25] focus on providing *source code level abstractions*: although the actual transformation is performed on bytecode, the API exposes concepts of the source language. This is highly profitable to end users. In its latest version [29], Javassist even offers a lightweight online compiler so that injected code can be specified as a string of source code. The Javassist compiler supports a number of dedicated metavariables, which can be used to refer to the context in which a piece of code is injected.

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

Proposals for Smalltalk. To the best of our knowledge there is no general-purpose bytecode manipulation tool for a Smalltalk dialect. AOStA [36] is a bytecode-to-bytecode translator that aims at providing higher-level, transparent, type-feedback-driven optimizations. It was not thought to be open to end users for bytecode manipulation [1]. Method wrappers [10] make it possible to wrap a method with before/after code. They are very fast to install and remove, as they do not need to parse bytecode or generate methods, but are not a general-purpose transformation tool. Several extensions actually need more power than just before/after control. AspectS [37] has been recently proposed as an aspect-oriented interface to the reflective capabilities of Smalltalk combined with method wrappers (to implement before/after advices). AspectS is actually a tool that would much profit from ByteSurgeon, as it would significantly raise its expressive power.

2.3 Motivation

From the above, it should be clear that a general-purpose bytecode manipulation tool for Smalltalk is missing. Such a tool ought to provide convenient abstractions to users, both at the source level and bytecode level. ByteSurgeon is precisely such a tool. Beyond its interest for the Smalltalk community, ByteSurgeon also opens the door to a brand new range of experiments with runtime bytecode transformations, since it has none of the limitations of existing Java proposals. For instance, ByteSurgeon makes it possible to analyze concrete issues of fully-dynamic AOP.

3 ByteSurgeon at Work

ByteSurgeon is our library for runtime program transformation in Smalltalk, currently implemented in the Squeak environment. ByteSurgeon complements the reflective abilities of Smalltalk [38] with the possibility to instru-

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

ment methods, down to method bodies. Smalltalk provides a great deal of structural reflection: the structure of the system is described in itself. Structural reflection can be used to obtain the object representing any language entity. For instance, the global variable Example stands for the class (the object representing the class) Example, and the object describing the compiled method aMethod in class Example is returned by the expression Example>>#aMethod. Dynamically adding instance variables and methods to an existing class is fully supported by any standard Smalltalk environment. However the structural description of a Smalltalk system stops at the level of methods: compiled method cannot be reflected upon. Conversely, ByteSurgeon can be used to do both introspection and intercession on compiled methods.

3.1 Introspecting Method Bodies

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

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

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

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

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

Instructions in a method body are static occurrences of the operations of a language. ByteSurgeon supports message send, access to instance variable and local variables, and constants. The structural model representing language operations is shown on Figure 1 [2]. This structural model is bytecode-based. It does not encode as much information as an AST does, *e.g.,* it is not possible to extract, from an IRSend, the instructions that correspond to the arguments of the send. This is a limitation of bytecode-based transformation against AST-based transformation.

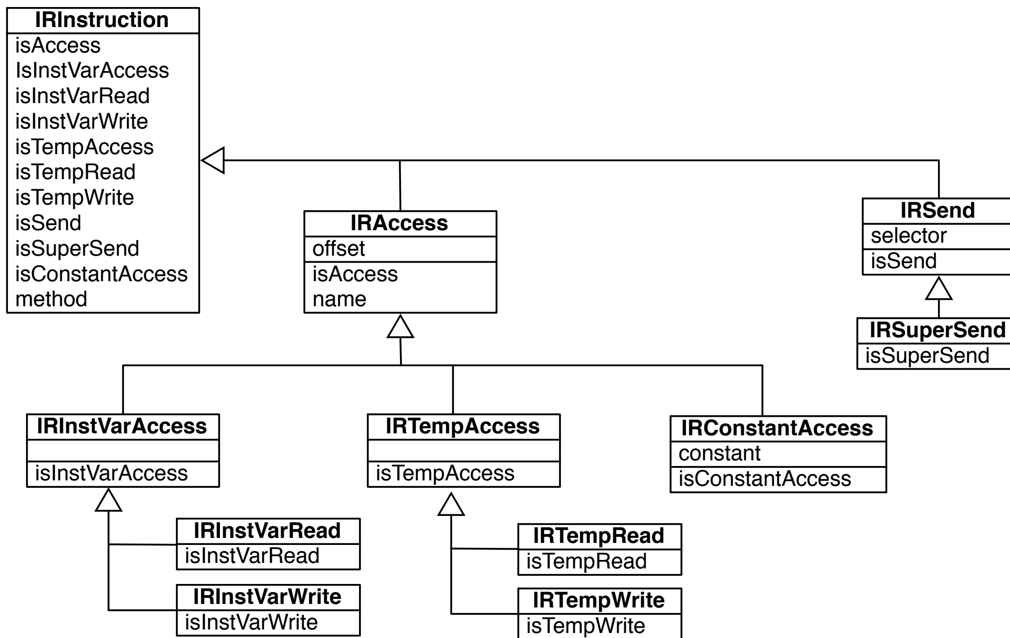

Fig. 1. Structural model of instructions in ByteSurgeon.

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

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

Method Evaluation. A peculiar language operation is *message receive* (the callee-side equivalent of a message send). Actually, a message receive is realized by two operations: method lookup and method evaluation. Since we

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

are working at the bytecode level, we do not have access to method lookup, only *method evaluation*. Rather than corresponding to a bytecode instruction inside a method body, method evaluation corresponds to a method body as a whole. Since ByteSurgeon treats all language operations in a uniform manner, methods have the same introspection and intercession interface than instructions (*e.g.,* see Section 3.3.3).

3.3 Modifying Method Bodies

ByteSurgeon supports two ways of modifying method bodies: a bytecode-level manner, where the user directly specifies the required transformation in terms of bytecode representations, and a source-level manner, where the transformation is specified with a string of source code. We hereby only present the source-level API. The bytecode-level API is briefly mentioned in Section 4.3.

Similarly to Javassist [29], ByteSurgeon provides an online compiler that makes it possible to specify code to be inserted as a string. The methods to insert code before, after and instead of an occurrence of a language operation are named respectively insertBefore:, insertAfter: and replace:. They take as argument the source code as a string, which is subsequently compiled by the ByteSurgeon compiler, and the resulting code is inserted at the appropriate position. For instance, the following code inserts a call to the system beeper before each message send occurring within Example>>#aMethod:

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

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

3.3.1 Accessing Static Information: User-defined Variables

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

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

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

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

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

3.3.2 Accessing Runtime Information: Metavariables

The online compiler of ByteSurgeon also supports a number of predefined metavariables that refer to information available at runtime, such as the receiver of a message send (Figure 2). Metavariables are an essential part of the expressiveness of a good bytecode transformation framework. The exact set of available metavariables depends on both the operation selected –in the case of a message send, metavariables are provided to refer to the sender, the receiver and the arguments– and the transformation to perform –when inserting after, it is possible to access the result–. Metavariables are denoted by the <meta: *#variable*> construct. For instance, the following code replaces each message send with a call to a dispatcher metaobject in charge of the actual method lookup [7, 39]:

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

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

3.3.3 Altering Method Evaluation

To support transformation of method evaluation, method objects also support the insertBefore:, insertAfter: and replace: messages. As an example, the

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

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

Fig. 2. Metavariables supported by ByteSurgeon.

following code inserts a trace before each evaluation of a method in Example:

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

The metavariables for method evaluation are the same as for message sending (see Figure 2). The following example uses a metavariable to access the method evaluation result:

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

4 Inside ByteSurgeon

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

4.1 Squeak

ByteSurgeon is currently implemented in Squeak [40], an open source implementation of Smalltalk-80 [41]. Squeak is based on a virtual machine that interprets bytecodes. During a normal compilation phase, method source code

is scanned and parsed, an abstract syntax tree (AST) is created and bytecodes are generated for the corresponding methods (Figure 3).

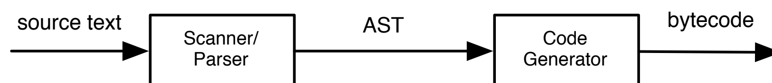

Fig. 3. The standard Smalltalk to bytecode compiler.

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

4.2 The Closure Compiler and its Intermediate Representation

The closure compiler [42] relies on a more complex bytecode generation step (Figure 4): first an *Intermediate Representation (IR)* is created; then the IR is used to generate the real bytecode (the raw numbers).

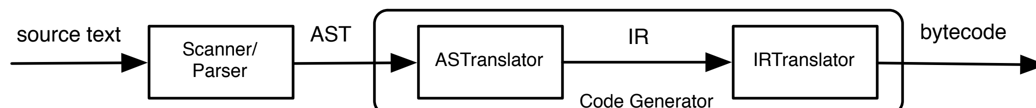

Fig. 4. The closure compiler.

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

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

As motivated in Section 2.2, ByteSurgeon ought to offer adequate abstractions for both bytecode-level and source-level transformations. The IR of the closure compiler actually represents an excellent alternative for working at the bytecode level: it makes it possible to express code that is not directly obtainable from Smalltalk source code, while abstracting away many details.

All classes reifying instructions (recall Figure 1) are from the closure compiler IR. The low-level transformation API of ByteSurgeon is based on these classes. In addition to the classes reifying instructions which correspond to language operations, the IR includes classes reifying bytecode-only instructions: IRPop, IRDup, IRJump, IRReturn, etc.

4.3 Low-level Transformation API

In Section 3, we have used the high-level API of ByteSurgeon to specify transformations giving a string of source code, which may contain metavariables to access dynamic information. The description of the new code to be inlined can also be done by directly editing the instruction objects for the IR hierarchy. In the following example, the selector of all sends of the message oldMessage:with: are replaced by sends of the message newMessage:with:, by using the selector: accessor of an IRSend object:

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

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

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

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

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

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

Apart from being slightly more verbose and relying on the use of the reflective message sending perform:with:, this approach requires the use of metavariables, which are more costly due to the associated preambles that needs to be generated (as shown in Section 4.4). Conversely, the low-level API makes it possible to do this transformation directly, without requiring runtime reification.

4.4 Implementation of Metavariables

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

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

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

```
77 pushConstant: 3
20 pushConstant: 4
B0 send: +
7C returnTop
```

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

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

To support metavariables, we need to add bytecode to store the necessary values, by popping them from the stack and storing them in additional temporary variables. In our example, we need the receiver. Since the receiver is deep in the stack, below the arguments, we also need to store the arguments

in temporary variables, to be able to access them afterwards. In the case of before/after, it is also necessary to rebuild the stack. The resulting bytecode for our example is as follows:

```
22 pushConstant: 3
23 pushConstant: 4
68 popIntoTemp: 0              "put argument in temp 0"
69 popIntoTemp: 1              "put receiver in temp 1"
24 pushLit: ##Transcript       "start of inserted code"
11 pushTemp: 1                 "push receiver for printing"
D5 send: asString
E6 send: show:
87 pop                         "end of inserted code"
11 pushTemp: 1                 "rebuild the stack"
10 pushTemp: 0
B0 send: +                     "original code"
7C returnTop
```

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

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

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

5 Validation

We now validate the interest of ByteSurgeon by showing how easy it is to implement two language extensions: method wrappers [10] and a simple runtime MOP for controlling accesses to instance variables. Section 5.3 completes this validation by reporting on performance measurements.

5.1 Method Wrappers

Method wrappers [10] wrap a method with before/after behavior. Wrapping a method is implemented by swapping out the compiled method by another one, valueWithReceiver:arguments: that calls the before method, then the original method, and finally the after method [5] :

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

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

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

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

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

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

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

[5] At the time of this writing, ByteSurgeon does not yet support exception handlers, so we actually implemented a simplified version where the after method is just inlined at the end of the method.

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

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

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

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

As we can see, method wrappers are straightforward to implement with ByteSurgeon. The complete implementation included in the distribution consists of 41 lines of code, with comments. This implementation of method wrapper should only serve as an example of use of ByteSurgeon, it is not meant to be a replacement yet since not all features of method wrappers are supported. Furthermore, as illustrated in Section 5.3, standard method wrappers and ByteSurgeon method wrappers have different performance profiles.

5.2 A Small Runtime MOP

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

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

TraceMO>>instVarStore: name in: object value: newVal
```

```
    Transcript show: 'var store: ', newVal printString; cr.
    ^object instVarNamed: name put: newVal.
```

This metaobject can be installed on class Point as follows:

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

The MOP>>install method uses ByteSurgeon to replace the bytecodes that read or store instance variables with calls to the metaobject (*aka.* hooks):

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

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

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

```
MOP uninstall: MOExample.
```

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

5.3 Benchmarks

We now report on several preliminary benchmarks[6] we have performed to evaluate the efficiency of ByteSurgeon. First, we report on transformation vs. compilation costs, and then study the performance of the standard implementation of method wrappers with that based on ByteSurgeon.

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

The first benchmark is applied to the Object class:

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

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

Class Object contains 429 methods, amounting to 2344 lines of code. We did the same experiment on a larger code base: the whole hierarchy of collection classes. This hierarchy consists of 76 classes, 2231 methods, summing up to 15783 lines of code. The benchmark is run as:

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

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

The results of both benchmarks are presented in Figure 5. As expected, ByteSurgeon performs very well. The highly optimized standard compiler is approximately twice slower than ByteSurgeon, while the new compiler, which is much easier to reuse and extend but less optimized, is around 6 times slower.

Method wrapper performance. We now compare the performance of the

[6] Machine used: Apple PowerBook 1.5Ghz, Squeak 3.8

	Object		Collections	
	time (ms)	factor	time (ms)	factor
ByteSurgeon	661	1	4817	1
standard compiler	1232	1.86	9760	2.03
closure compiler	3673	5.55	33611	6.98

Fig. 5. Comparing compilation and transformation times.

standard implementation of method wrappers with that based on ByteSurgeon. We compare both installation (transformation) time and execution time.

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

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

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

```
[1000 timesRepeat: [
  w := TestMethodWrapper on: #run inClass: Bench.
  w install. w uninstall]] timeToRun
```

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

```
Bench>>run
   | t |
   BCounter inc.
   t := 3+4.
   BCounter inc.
   ^t.
```

To be fair in our evaluation, we changed the execution semantics of standard method wrappers, so that they do not wrap the after in an exception handler, but rather inline both before and after methods. The benchmark for both cases is run as follows:

```
[1000000 timesRepeat: [Bench new run]] timeToRun
```

The results of the benchmarks (Figure 6) show that ByteSurgeon is slower

Method Wrapper implementation	Installation time (ms)	Installation factor	Runtime time (ms)	Runtime factor
Hand-coded	–	–	1253	1
Standard	603	1	6732	5.37
ByteSurgeon	3710	6.01	1222	0.98

Fig. 6. Comparing installation and runtime performance of method wrapper implementations.

for installing wrappers. This was expected because method wrappers actually simply swap the wrapped compiled method with the wrapper one, while ByteSurgeon actually modifies the original method. The other side of the coin is that ByteSurgeon-based method wrappers are much more efficient at runtime. Standard method wrappers are 3.5 times slower than the hand-coded version, while the ByteSurgeon implementation is as fast as the hand-coded version. The slight enhancement that can be observed comes from the fact that, in the considered case, ByteSurgeon does not need to use a temporary variable to store the return value, it just uses the stack.

6 Conclusion and Future Work

We have presented ByteSurgeon, an efficient library for runtime bytecode manipulation in Smalltalk, implemented in Squeak. We have shown:

- APIs for specifying transformations that allow users to control the tradeoff between expressiveness and performance for the code to be inlined: ByteSurgeon users can either specify Smalltalk code with metavariables or specify the code at the bytecode level.
- the expressiveness of ByteSurgeon by showing how well-known language extensions are concisely expressed, and reported on preliminary benchmarks validating our efficiency claim.
- the runtime capabilities of ByteSurgeon with a simple MOP that can be dynamically installed and retracted. Such runtime changes are not feasible in a static system like Java without changing the virtual machine.

Future work can be dividing in two directions: the first is to continue improving ByteSurgeon as such, and the second consists in using ByteSurgeon in a number of projects that will directly benefit from its features. Of course, both tracks mutually benefit from each other.

Regarding ByteSurgeon itself, there is a number of features that are being discussed at this time. In particular, ByteSurgeon should be extended with

support for exception handling. It is also appealing to offer a kind of proceed instruction to trigger the execution of a replaced operation occurrence from inside the metacomputation. Another direction to explore is that of the abstraction layer used to describe a method. As of now we use a bytecode representation, but it would be interesting to explore the direct use of abstract syntax trees at this level. The choice between AST and bytecode presents a tradeoff between performance and expressiveness: decompiling to AST and code-generation will be slower than using the the bytecode-level abstractions of the IR, but in turn we gain a lot in expressiveness and ease of use, since AST is more structured than IR. We plan to explore these tradeoffs in the future.

As regards applications of ByteSurgeon in other projects, the perspectives are manifold. We plan to use ByteSurgeon for code annotation to collect runtime traces of program execution to support *omniscient debugging* [11]. *Reflex* is a system based on bytecode transformation providing partial behavioral reflection in Java [43]. It has recently evolved to a versatile kernel for multi-language AOP [44], easing the implementation of (domain-specific) aspect languages and providing support for the detection and resolution of aspect interactions. The on-going *Geppetto* project aims at exploring the possibilities offered by an implementation of Reflex in Squeak, using ByteSurgeon, enjoying the flexibility of true runtime code transformation.

Acknowledgements. We thank David Röthlisberger and the anonymous reviewers for their comments.

References

[1] R. Keller, U. Hölzle, Binary component adaptation, in: ECOOP'98, LNCS 1445, 1998, pp. 307–340.

[2] D. L. Parnas, On the criteria to be used in decomposing systems into modules, CACM 15 (12) (1972) 1053–1058.

[3] R. Stroud, Z. Wue, Using metaobject protocols to satisfy non-functional requirements, in: Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press, 1996, pp. 31–52.

[4] É. Tanter, J. Piquer, Managing references upon object migration: Applying separation of concerns, in: Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001) (jan 2001).

[5] J. McAffer, Meta-level architecture support for distributed objects, in: Proceesings of the Fourth International Workshop on Object-Orientation in Operating Systems, 1995., 1995, pp. 232–241.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: M. Aksit, S. Matsuoka (Eds.), Proceedings ECOOP '97, Vol. 1241 of LNCS, Springer-Verlag, Jyvaskyla, Finland, 1997, pp. 220–242.

[7] S. Ducasse, Evaluating message passing control techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP) 12 (6) (1999) 39–44.

[8] J. H. Heinz-Dieter Bocker, What tracers are made of, in: Proceedings of OOPSLA/ECOOP '90, 1990, pp. 89–99.

[9] F. Pachet, F. Wolinski, S. Giroux, Spying as an Object-Oriented Programming Paradigm, in: Proceedings of TOOLS EUROPE '93, 1993, pp. 109–118.

[10] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the Rescue, in: Proceedings ECOOP '98, Vol. 1445 of LNCS, Springer-Verlag, 1998, pp. 396–417.

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

[12] A. J. Ko, B. A. Myers, Designing the whyline: a debugging interface for asking questions about program behavior, in: Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems, Vol. 1, 2004, pp. 151–158.

[13] A. H. Borning, D. H. Ingalls, Multiple inheritance in Smalltalk-80, in: Proceedings at the National Conference on AI, Pittsburgh, PA, 1982, pp. 234–237.

[14] B. Garbinato, R. Guerraoui, K. R. Mazouni, Distributed programming in GARF, in: R. Guerraoui, O. Nierstrasz, M. Riveill (Eds.), Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Vol. 791 of LNCS, Springer-Verlag, 1994, pp. 225–239.

[15] J. K. Bennett, The design and implementation of distributed Smalltalk, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 318–330.

[16] P. L. McCullough, Transparent forwarding: First steps, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 331–341.

[17] K. Beck, Instance specific behavior: Digitalk implementation and the deep meaning of it all, Smalltalk Report 2(7).

[18] J.-P. Briot, Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment, in: S. Cook (Ed.), Proceedings ECOOP '89, Cambridge University Press, Nottingham, 1989, pp. 109–129.

[19] Y. Yokote, M. Tokoro, Experience and evolution of ConcurrentSmalltalk, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 406–415.

[20] G. A. Pascoe, Encapsulators: A new software paradigm in Smalltalk-80, in: Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, 1986, pp. 341–346.

[21] B. Foote, R. E. Johnson, Reflective facilities in Smalltalk-80, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 327–336.

[22] J. McAffer, Meta-level programming with coda, in: W. Olthoff (Ed.), Proceedings ECOOP '95, Vol. 952 of LNCS, Springer-Verlag, Aarhus, Denmark, 1995, pp. 190–214.

[23] W. R. LaLonde, M. V. Gulik, Building a backtracking facility in Smalltalk without kernel support, in: Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, 1988, pp. 105–122.

[24] G. Kiczales, J. des Rivières, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[25] É. Tanter, M. Ségura-Devillechaise, J. Noyé, J. Piquer, Altering Java semantics via bytecode manipulation, in: Proceedings of GPCE'02, Vol. 2487 of LNCS, Springer-Verlag, 2002, pp. 283–89.

[26] M. Tatsubori, S. Chiba, M.-O. Killijian, K. Itano, OpenJava: A class-based macro system for java, in: 1st OOPSLA Workshop on Reflection and Software Engineering, Vol. 1826 of LNCS, Springer Verlag, 2000, pp. 117–133.

[27] J. Bachrach, K. Playford, The Java Syntactic Extender (JSE), Proceedings of OOPSLA '01, ACM SIGPLAN Notices 36 (11) (2001) 31–42.

[28] M. Dahm, Byte code engineering, in: Proceedings of JIT '99, Düsseldorf, Deutschland, 1999, pp. 267–277.

[29] S. Chiba, M. Nishizawa, An easy-to-use toolkit for efficient Java bytecode translators, in: Proceedings of GPCE'03, Vol. 2830 of LNCS, 2003, pp. 364–376.

[30] D. A. Smith, A. Kay, A. Raab, D. P. Reed, Croquet, A Collaboration System Architecture, in: Proceedings of the First Conference on Creating, Connecting and Collaborating through Computing (2003).

[31] Jython, http://www.jython.org/.

[32] Java debug interface (jdi), http://java.sun.com/j2se/1.4.2/docs/jguide/jpda/jarchitecture.html.

[33] S. Liang, G. Bracha, Dynamic class loading in the Java virtual machine, in: Proceedings of OOPSLA '98, ACM SIGPLAN Notices, 1998, pp. 36–44.

[34] E. Bruneton, R. Lenglet, T. Coupaye, ASM: A code manipulation tool to implement adaptable systems, in: Proceedings of Adaptable and extensible component systems (nov 2002).

[35] S. Chiba, Load-time structural reflection in Java, in: Proceedings of ECOOP 2000, Vol. 1850 of LNCS, 2000, pp. 313–336.

[36] E. Miranda, A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk, unpublished (2002).

[37] R. Hirschfeld, AspectS – Aspect-Oriented Programming with Squeak, in: M. Aksit, M. Mezini, R. Unland (Eds.), Objects, Components, Architectures, Services, and Applications for a Networked World, no. 2591 in LNCS, Springer, 2003, pp. 216–232.

[38] F. Rivard, Smalltalk : a Reflective Language, in: Proceedings of REFLECTION '96, 1996, pp. 21–38.

[39] J. Ferber, Computational reflection in class-based object-oriented languages, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 317–326.

[40] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press, 1997, pp. 318–326.

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

[42] A. Hannan, Squeak Closure Compiler, http://minnow.cc.gatech.edu/squeak/ClosureCompiler.

[43] É. Tanter, J. Noyé, D. Caromel, P. Cointe, Partial behavioral reflection: Spatial and temporal selection of reification, in: Proceedings of OOPSLA '03, ACM SIGPLAN Notices, 2003, pp. 27–46.

[44] É. Tanter, J. Noyé, A versatile kernel for multi-language AOP, in: Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005), Vol. 3676 of LNCS, Tallin, Estonia, 2005.