SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 01; **34**:1–12 *Prepared using* speauth.cls *[Version: 2000/03/06 v2.1]*

An architecture for safe bytecode insertion

Geoff A. Cohen* and Jeffrey S. Chase

Department of Computer Science
Duke University
{gac,chase}@cs.duke.edu

SUMMARY

Bytecode transformation is a powerful technique to dynamically extend or adapt Java software using *transformer* programs that automatically rewrite compiled classes. Bytecode transformers have been used to integrate separately developed software components, instrument programs with monitoring or debugging code, and extend programs with new features for security, data management, and other needs. Although bytecode transformation has shown promise in research systems, acceptance of this technique in practice depends on effective software tools to facilitate development of transformers that are effective and correct across all legal inputs.

This paper explores interfaces for safe code insertion in JOIE, a toolkit for constructing bytecode transformers for the Java environment. JOIE provides a core set of low-level primitives for manipulating Java classfiles, supplemented by higher-level interfaces that directly support common transformer styles. The goal of this design is to make common transformations safe and efficient, without sacrificing the toolkit's generality for more complex transformers.

JOIE's high-level facilities include expressive interfaces for safe insertion of new code into existing applications, which is the focus of this paper. A key technique is to extend compiled classes by *mixing in* attributes, behaviors, and interfaces from auxiliary classes that are written in Java and packaged with the transformer. In addition the JOIE architecture supports modular primitives for traversing and filtering existing bytecode and splicing in new instruction sequences. This paper presents the architecture of JOIE's code insertion facilities, and illustrates their use in three example transformations. These examples expose issues and tradeoffs in the design and use of the interfaces, and demonstrate the potential of the JOIE bytecode transformation architecture as a tool for software extension.

KEY WORDS: Java; bytecode; program transformation; architecture; interfaces

1. Introduction

Bytecode transformation [3] is a powerful technique for automatically adding functionality to Java programs and other software packaged to run in the Java environment [9]. In this technique, *transformer* programs modify existing software by automatically parsing and manipulating compiled classes represented as *classfiles.* Program transformation is valuable primarily for implementing general features and adaptations that may be specified independently of the application functionality. The new features are decoupled from the application programs and implemented once — in the transformer — rather than reimplemented in each program. Because they operate on the compiled program, bytecode transformers allow late adaptations or extensions of packaged software, even if source code is not available.

Researchers have used bytecode transformation for a wide range of purposes including resource accounting [4], performance instrumentation [15], distributed verification [18], component adaptation [11], and support for generic types [1]. These experiments show the potential of bytecode transformation as a basic technique for creating richer, more full-featured software systems, particularly in networked systems where the transportable nature of Java bytecode makes the Java programming environment most attractive.

Despite the success of these and other research systems, there are several critical obstacles to the acceptance of bytecode transformation as a software development tool. Constructing transformers requires a detailed knowledge of the Java classfile representation and bytecode instruction set, and disciplined programming to ensure that the transformer is safe and correct for a full range of input classfiles. Transformer errors may remain hidden until the transformer encounters specific uncommon input conditions that trigger them. Some errors manifest only when the transformed class executes, making them difficult to isolate.

This paper explores facilities for safe bytecode insertion in JOIE, a bytecode transformation toolkit. We built JOIE both as a software development tool and as a vehicle to explore the power and limitations of bytecode transformation. Previous versions of JOIE [3] have been used by the research community to construct a number of systems based on bytecode transformation, including novel security architectures [23] [5], meta-object protocols [24], automatic replication [2], a speech-enabled user interface [6], and automated testing [20]. The experience gained from these efforts has guided enhancements to the JOIE architecture. The purpose of this paper is to explore the design principles and architectural features of JOIE that enable these and other applications, focusing on the bytecode insertion primitives used by JOIE bytecode transformers to extend application functionality.

JOIE provides a high-level programming interface for developing bytecode transformers in the Java language. Butler Lampson and others have pointed out that interface design is the most important challenge of system design [13] and often the most difficult, due to the tension between the goals of simplicity, completeness, and efficient implementation. For JOIE, the challenge is to provide an interface that is expressive enough to allow a full range of uses of bytecode transformation, yet constrained enough to be safe, easy to use, and efficient to implement. The JOIE architecture resolves these inherent conflicts by offering a set of high-level interfaces to simplify common styles of code insertion, layered above unrestricted low-level interfaces for editing classfiles. The high-level interfaces offer a tradeoff between power and ease of use; constraining the possible changes enables them to define specialized groups of transformations easily and safely. The low-level interfaces offer more power and more control over transformer behavior, but they can be difficult and dangerous to use directly. Transformers may use these interfaces alone or in combination to implement simple or complex transformations. While JOIE exposes the low-level interfaces to transformer authors for completeness, the rich high-level interfaces render them unnecessary for most applications.

This paper makes the following contributions:

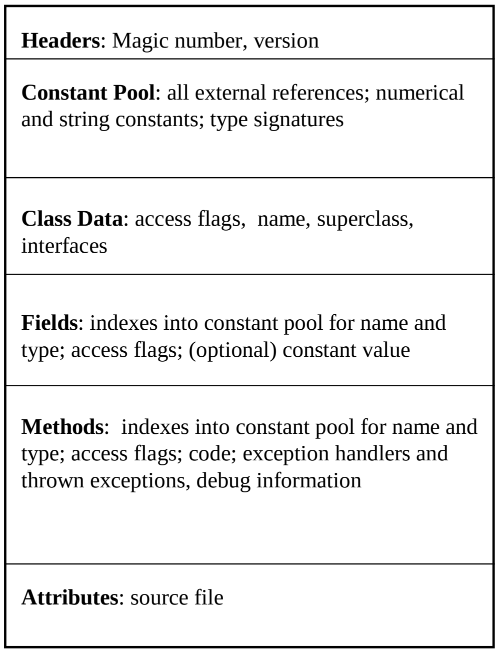
- It gives an overview of the nature and role of bytecode transformation in the Java development environment, using the JOIE transformation architecture to illustrate key issues.
- It identifies common transformer styles, and shows how JOIE supports these styles with high-level code insertion interfaces that are expressive, safe, easy to use, and well-matched to common transformer needs. These interfaces include a SYMBOLIC interface to modify or synthesize instruction sequences, a MIXIN interface to extend target classes by injecting attributes from auxiliary classes written in Java, and an INSTRUMENTATION interface to select sites for code insertion using modular traversal and filtering primitives.
- It illustrates the power and limits of the code insertion interfaces by presenting representative examples of JOIE bytecode transformations for security and data management. Each example is a useful building block for enhancing functionality of Java applications. Together they demonstrate the range of uses of bytecode transformers and their implementation with JOIE.

This paper is organized as follows. Section 2 gives an overview of the JOIE toolkit, relevant aspects of the Java bytecode environment, and the role of bytecode transformation. Section 3 presents the JOIE code insertion architecture and its rationale, and outlines the key properties of the selected interfaces. Section 4 explores the use of the JOIE transformer interfaces for the example applications. Section 5 concludes.

2. Overview of Bytecode Transformation with JOIE

JOIE transformers are Java classes that use the JOIE programming interfaces to examine and manipulate the target classes. Transformers operate on input classes represented in their compiled, transportable form as *classfiles*. The classfile representation is defined by the Java Virtual Machine (JVM) specification [16], and is the basic format used to store Java software in *.class* and *.jar* files and to transmit classes across a network. During execution, the JVM dynamically imports classfiles into the runtime environment to resolve references to classes that have not yet been loaded. To obtain a referenced class, the JVM passes the name to a *classloader* object in the local environment, which locates and fetches the classfile from the file system or from a network server. The JVM then parses the classfile and converts it to an instance of `java.lang.Class`, the runtime representation of a Java class.

JOIE supports transformer authors through its interfaces to examine and manipulate the contents of classfiles. The JOIE interfaces include primitives to access most elements of the Java classfile; some of these primitives are similar to the Java reflection API. Figure 1 illustrates the structure of a classfile, which contains six main sections: a *header* including a magic number and Java version number; the *constant pool*; class data including the name, superclass, access permissions, and the interfaces it implements; a list of fields (the member data of the class



Headers: Magic number, version

Constant Pool: all external references; numerical and string constants; type signatures

Class Data: access flags, name, superclass, interfaces

Fields: indexes into constant pool for name and type; access flags; (optional) constant value

Methods: indexes into constant pool for name and type; access flags; code; exception handlers and thrown exceptions, debug information

Attributes: source file

Figure 1. Structure of a Java classfile

and of objects that are instances of the class); a list of methods (or functions) and their implementation; and a list of attributes, including the name of the source file from which the class was compiled. The constant pool is a list of fixed- and variably-sized entries that contain the names and type signatures of all the classes, methods, fields, and constant values (including strings) defined or referenced within the class. Every reference within the class is represented by an index into the constant pool.

A JOIE transformer may examine or modify any section of the classfile. For example, it might add methods or insert instructions into an existing method, edit class data to add a new field to a class, change the type of an existing field, expose private interfaces, add new interfaces, or change the superclass. A transformer may add entries to the constant pool if, for example, it inserts instructions that reference new classes, fields, methods, or constants.

Java classfile methods are composed of *bytecodes*, which are the JVM instructions that define the functionality of the method. Each bytecode contains a one-byte operation code (the "form" of the bytecode) optionally followed by arguments. Any instruction that uses a reference to a method, class, or field represents that reference by a sixteen-bit index into the constant pool.[†]

[†]One exception, `ldc`, uses an eight-bit index to load a constant from the constant pool.

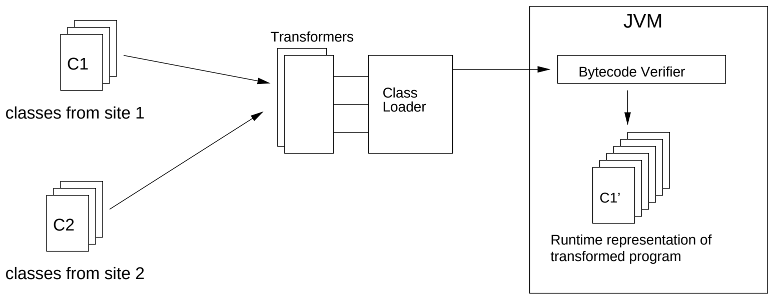


Figure 2. Transforming classes at load time

Using the JOIE interfaces, the transformer may iterate through the instructions of a method, insert or remove instructions at arbitrary points, or modify existing instructions.

The JVM is conceptually a stack machine; all instructions are stack operations. For example, an `iadd` instruction consumes two integer operands from the top of the stack, adds them, and pushes the result onto the stack. JVM instructions may also directly address the *frame*, a fixed-size array of logical registers for local variables and method parameters. The nature of the JVM simplifies some aspects of bytecode transformation and complicates others, as discussed in later sections.

2.1. When to Transform

Bytecode transformation is an example of late code modification [21], which targets an executable program form after compilation. Although late code modification loses access to useful information available only in the source code, classfiles in the Java environment retain a great deal of symbolic information in the constant pool, enabling a wide range of transformations. Transformers may be selected and applied statically after compilation, or dynamically at load time as classes are imported [3], when more is known about the environment in which the code is to run. In a networked environment with transportable code, load-time transformations may be applied at the server as specific clients request the class, at an intermediate proxy that transforms classes as they pass through the network, or by the classloader as the classes enter the JVM (see Figure 2). Load-time transformation can improve reuse of existing code by providing a means to adapt it to the target environment.

The JOIE toolkit supports load-time transformation directly with an extended classloader that automatically applies registered transformations to each class brought into the local environment. This form of load-time transformation occurs before the JVM has access to the class, thus it is transparent to the JVM and does not compromise or interfere with subsequent class operations by the JVM. These operations include *verification* to ensure that the transformed class is semantically valid, and (in some JVMs) *just-in-time compilation* to convert executed code to native instructions for the underlying machine.

2.2. What to Transform

JOIE enables *whole-program* transformation to add features to every class of an application, or to verify that none of the classes uses restricted features or interfaces. *Static* whole-program transformation assumes that all needed classes are present at transformation time. This "closed world" assumption enables a range of inter-method and inter-class analysis and optimization. *Dynamic* (load-time) transformation enables whole-program transformations of software imported from multiple sources on the network. However, although it ultimately covers the entire program, load-time transformation is *incremental*; a load-time transformer must process some classes before other classes are available. The transformer does not control the load order of the classes, and most current JVMs do not allow modification or reloading of classes once they are loaded. Transformers thus have only one chance to transform classes, and often must do so with incomplete information about the entire application. This constraint can limit the applications for load-time transformation. For example, a dynamic transformer cannot in general apply optimizations requiring interprocedural analysis (such as leaf method removal), because the transformation may operate on a call site before the matching method is loaded. Transformer control over the class load order would improve the generality of dynamic transformation.

A *selective* transformation targets a specific subset of application classes. Selective transformers may act as development tools to automatically expand specified target classes, saving programmers from adding boilerplate code manually. The targets for a selective transformer might be listed in a configuration file, or the transformer can infer the targets from inspection of other classes that reference them. One useful alternative is to define a common *labeling interface* whose name is known to the transformer; application programmers may designate target classes — or subsets of their methods — by marking them as implementing that interface. The transformer then triggers to select the classes and/or methods that are associated with the labeling interface.

One purpose of selective transformers is to extend or expand the interfaces (or fields) in the target classes. Other source code in the application may access the modified interfaces directly, but only if the transformer is applied statically and the program construction process applies it before compiling any classes that depend on its effects. If this is not possible, then the application may access the new features using an *interface wrapper*. For example, to invoke a method that has not yet been added to a class, the application code may instead invoke an associated static wrapper method, passing the object as an argument. At runtime, the wrapper casts the object as needed to access the new interface, leaving the JVM to check the type at runtime and throw an exception if the object does not actually implement the interface.

2.3. Structure of JOIE Transformers

JOIE incrementally parses each input classfile into an intermediate representation (IR) accessible through the class `joie.ClassInfo`. Transformers invoke methods of `ClassInfo` and related JOIE classes to access and update the IR. When the transformation is complete, JOIE regenerates a valid classfile from the transformed IR.

Transformers are implemented as Java classes that implement the JOIE interface `joie.ClassTransformer`. Before beginning to parse a class, the JOIE environment calls `select(String name)` on registered transformers, passing the fully qualified class name as an argument. JOIE applies a transformer to the class only if the transformer's `select` method returns true. JOIE passes only the class name because it is all that is known about the class until the classfile is selected and parsed. If `select` returns true, JOIE invokes the transformer's `transform(ClassInfo cinfo)` method, passing the `ClassInfo` for the selected class. In `transform`, the transformer may request any information from the classfile. A selective transformer might choose to proceed only if the class meets some set of criteria, such as implementing a given interface, subclassing a given class, or containing a certain field or method.

The transformer classes implement the logic to edit and manipulate the target classfiles. As part of that manipulation, a transformer may inject new functionality into the target. While the transformer may synthesize new bytecodes on the fly, it is often easier and safer to write the new code fragments as *auxiliary classes* in Java, compile them, and include them as data for the transformer. Auxiliary classes do not appear by name in the final running program; rather, the transformer draws material from them to *splice* code and fields into the target classes. For example, the JOIE MIXIN interface (see Subsection 3.3) is based on this technique. Note also that the new code may invoke external support classes; for example, JOIE transformers in the Ivory system [2] inject calls to a package of data replication primitives.

There are several ways to compose transformers in JOIE to create more sophisticated transformations. Groups of simple transformers may be applied to the same class in sequence. For example, the JOIE classloader applies all registered transformers to each `ClassInfo` in registration order, amortizing the cost of parsing, analyzing, and reserializing the class across multiple transformations. Alternatively, transformer authors may create complex transformations by drawing on existing transformer code or packages of transformer primitives layered above JOIE. Since JOIE transformers are themselves Java classes, existing transformers may be subclassed, called by other transformers, or parameterized to select different targets or otherwise behave differently.

2.4. Related Work

Several other systems offer or use bytecode transformation or related code rewriting techniques as a basis for software development or extension.

Late code modification, also called executable editing, has been an important tool in the systems community since the early 1990s [21]. Tools exist to edit executable images for the Digital Unix/Alpha architecture [19], SPARC [14], and Win32 [17]. These are distinct from bytecode transformation in a number of ways. There are significant differences between the amount and type of information available in the executable images and bytecodes; bytecodes contain more symbolic data, retain the object-oriented nature of the class, and allow comparatively simple insertion of instructions. These architectural features of the JVM make bytecode transformation particularly suited to post-compilation changes to programs.

BIT [15] offers an instrumentation interface for Java classfiles modeled on ATOM [19]. Like ATOM, it constrains the transformer interface to preserve the semantics of the instrumented code. For example, a BIT code splice is constrained to a static method call with a single argument. While the BIT interface is powerful enough for many important applications of bytecode transformation, JOIE provides similar safety properties for simple transformers without limiting the power of the applications employing it.

Binary Component Adaptation [11] is a bytecode transformation environment designed to solve the problem of integrating incompatible software components. It addresses external interface issues, such as method signatures and names, but not implementations. Section 4.1 discusses using code insertion to alter the interfaces of target classes.

JRes [4] and Kimera [18] use bytecode transformation as tools for systems programming, inserting resource accounting calls and distributed virtual machine infrastructures (such as verification) respectively. Neither is represented as a general-purpose transformation environment, concentrating on the application. Either could in principle be run using JOIE as the transformation engine. These applications operate by inserting new functionality into the bodies of existing methods; see Section 4.2, which discusses method modification.

Safkasi [23] and Naccio [5] use JOIE to transform Java classes for alternative security architectures. Like JRes and Kimera, they do not themselves offer a programmable transformation interface as JOIE does. Safkasi and Naccio make changes at call sites in the target application; see Section 4.3 for a further discussion of call site modification.

AspectJ [12] is a source language extension for Java allowing the separate development of applications and *aspects* of code encapsulating features that cross-cut the existing methods. Like JOIE's code insertion interfaces, AspectJ allows programmers to apply these aspects to existing applications, producing composed applications that contain the aspect logic. The two projects are similar in spirit, but solve different problems. AspectJ concentrates more on the language model, adding new keywords and carefully specifying the semantics of composition. In contrast, JOIE concentrates on bytecode transformation as the *mechanism* of composition. The two approaches are thus complementary. The most important difference between the two is that JOIE encourages a strong separation between the composition logic, embodied in the transformer, and the new application logic, embodied in auxiliary classes. AspectJ aspects contain both the specification of the aspect logic and the composition rules for where that logic is inserted.

3. Bytecode Transformation Interfaces

This section outlines the design goals for the JOIE bytecode transformation toolkit, presents an overview of the toolkit architecture developed to meet those goals, and explores the key elements of the architecture in detail.

The JOIE toolkit interface is made up of a collection of classes supporting different views and operations on the underlying class representation. The goals of the JOIE toolkit and its API are common to any language design problem:

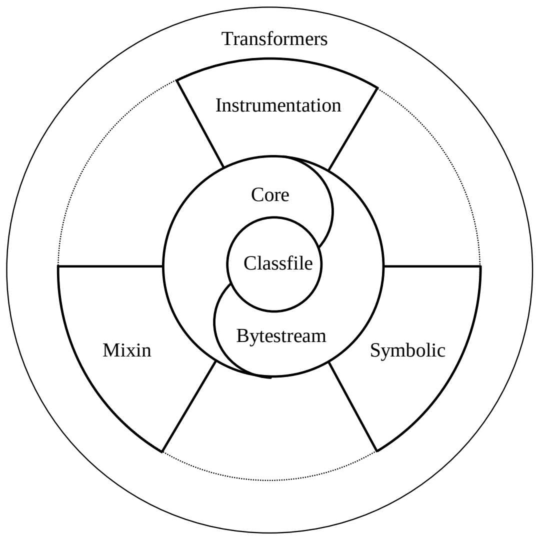


Figure 3. Relationships between interfaces.

- *Ease of Use.* The API should allow authors to construct simple transformers without mastering a complex interface or understanding specific details of the classfile representation.
- *Expressiveness.* The API should allow transformers to express the full range of useful transformations directly and concisely.
- *Performance.* Transformers should execute quickly and produce output classes that execute quickly. Although JOIE has features to produce faster transformed bytecode, transformation speed is also important because load-time transformers execute in the critical path of the application.
- *Correctness and Safety.* The toolkit should guide transformer authors to express transformations that are safe and correct. Although the toolkit cannot always detect an unsafe transformer, prevent it from damaging classes, or even ensure that it does what its author intended, the API should make it easy to express safe transformers and difficult to express unsafe ones. Also, the API methods should maintain consistency of the underlying class representation automatically when possible, while minimizing unexpected side effects.

3.1. Overview of the JOIE Architecture

JOIE meets its design goals through a nested structure, with specialized high-level "outer" interfaces layered above an "inner" core of low-level primitives. Figure 3 depicts the interfaces and their relationships. The center circle in Figure 3 represents the actual classfile, accessed only through the inner CORE and BYTESTREAM interfaces described below. The inner primitives are simple and maximally expressive; although they are sufficiently powerful to express any valid transformation, using them directly is difficult and error-prone. The outer interfaces simplify transformer development by offering high-level functionality specialized for common transformer styles. A transformer author may select the interface best suited for a given application, or use multiple outer interfaces in concert as a powerful way to express more complex transformations. At the same time, the API exposes the inner primitives to transformers with unanticipated needs, in keeping with Lampson's interface design axiom "don't hide power" [13]. In this way JOIE meets the competing goals of expressiveness on the one hand and safety and ease-of-use on the other.

The outer interfaces allow transformers to specify important operations using a few simple primitives that encapsulate the low-level details, while discouraging expression of unsafe transformations. This paper explores three high-level interfaces included in Figure 3:

- The MIXIN interface allows transformers to access and manipulate class-level structures, such as fields and methods, modifying them to incorporate features from transformer-supplied auxiliary classes. This provides a simple way to add fields, methods, or method prologues and epilogues without requiring knowledge of JVM bytecodes.
- The INSTRUMENTATION interface allows a transformer to insert code *splices* — injected sequences of instructions — at well-defined points of methods. The transformer may specify the traversal used to visit the method instructions, define filters to select insertion points, and supply methods to dynamically generate the spliced code from surrounding context.
- The SYMBOLIC interface is designed to create and manipulate instructions. It represents instructions as instances of classes that manage much of the bookkeeping complexity and insulate the transformer author from subtle distinctions among similar instructions. The symbolic interface interprets instruction operands and presents them to the transformer as references to typed objects with useful methods and behavior.

3.2. Core and Bytestream interfaces

Before presenting the high-level transformer interfaces, we first discuss the low-level CORE and BYTESTREAM interfaces on which they are based. These inner interfaces offer raw, direct, and unchecked access to the classfile structures for class content and bookkeeping, as described in Section 2. They deal directly with low-level data types including symbolic strings and unsigned bytes, and integers representing classfile table indices, offsets into byte arrays, mappings from numbers to types, or literal values. Together they allow a transformer author to read and update instructions and entries in the constant pool, manipulate instructions, or otherwise examine and change almost any byte of the classfile.

```
Method aMethod = cinfo.getMethod("serialize");
Code code = aMethod.getCode();
short cpool_index = cinfo.getConstantPool().getMethodrefIndex
          ("IvoryPackage", "serialCounter", "()V");
Instruction inst = new Instruction_2
          (Instruction.INVOKESTATIC, cpool_index);
code.prefix(inst);
```

Figure 4. Using BYTESTREAM to insert a method prefix splice.

```
Method[] methods = cinfo.getMethods();
for(int i=0; i<methods.length; i++) {
  CodeIterator iter = methods[i].getCode().getCodeIterator();
  while(iter.hasNext()) {
    Instruction instruction = iter.nextInstruction();
    System.out.println(instruction.getString());
  }
}
```

Figure 5. A simple disassembler implemented using CORE.

BYTESTREAM allows the caller to operate on instructions. BYTESTREAM represents embedded references to methods, fields, or classes as 16-bit indices into the constant pool; the referenced entry consists of two 16-bit indices referencing the constant pool entries for the containing class and the type or signature of the field or method. Figure 4 illustrates some of the BYTESTREAM methods. Note that references to methods are represented by a short integer obtained from the `getMethodrefIndex` method (part of the CORE interface). In the BYTESTREAM interface, a base `Instruction` class represents a single JVM instruction with no operands, such as math operations or "short-form" frame references. Subclasses for single-operand instructions are named `Instruction_`N, where N is the number of bytes in the operand. Other subclasses of `Instruction` represent the few instructions with multiple or a variable number of operands, including table switches, multidimensional object array creation, and interface method invocation.

CORE allows the caller to enumerate the fields and methods of a class, insert new fields, methods, or instructions, create entries in the constant pool, and get and set type information such as the name of the class, the superclass, and the class interfaces. BYTESTREAM and CORE can be used in concert to insert new instruction splices into existing methods. Figure 5 illustrates with an example use of the CORE interface.

While these low-level interfaces are powerful and efficient, they are unsuited to direct use by transformer authors because they require detailed knowledge of the classfile specification. Even with such knowledge, transformer authors may be forced to combine a large number of order-dependent CORE and BYTESTREAM primitives to achieve a desired result. The power of these interfaces also makes them unsafe: although JOIE automatically generates some fields such as structure sizes as it reserializes each class, classes modified using Core and Bytestream may be malformed or incorrect in a variety of ways. For example, since there is no compile-time or transformation-time type-checking, transformers using Core and Bytestream may encode constant pool offsets that are rendered incorrect or out-of-range by later insertions or deletions in the constant pool.

The following subsections describe how JOIE overcomes these limitations with specialized high-level interfaces built above Core and Bytestream.

3.3. Mixin Interface

The Mixin interface allows a transformer to mix the elements of an auxiliary class (see Section 2.3) into the target class. The transformer may use Mixin to introduce fields, methods, and supported interfaces to the target classes, as illustrated in Figure 6. In principle this is similar to multiple inheritance, but the resulting class does not share the type of the auxiliary class.

Mixin is also a powerful and easy-to-use interface for code splices. Code splices using Mixin need not directly inspect or modify the instructions of a method. Instead, a Mixin splice draws code from an auxiliary method. The advantage of this technique — called a *method splice* — is that the inserted code may be written in Java and compiled in the usual fashion, thus method splices are safer and easier to use than splices synthesized directly as a set of bytecode instructions. Mixin transformers may use method splices to add methods to the target class, or to use method code as prefixes or suffixes of existing methods of the target class. Method splices are defined in a subclass of the auxiliary class, and may refer to fields or methods of the parent class mixed into the target.

The source method for a method splice must conform to a set of easily checked constraints. For example, method splices must return `void`. In addition, they may not:

- take parameters,
- leave any items on the stack,
- contain any backward branches,
- throw or catch any checked exceptions,
- access any fields defined by the target class, or
- call methods defined by the target class.

These restrictions are sufficient to ensure that method splices are compatible with existing code. The splice cannot affect the state of the computation of the target method because it leaves the stack unchanged, and it cannot access the local variables or fields of the target class. The splice cannot stop execution of the target method because it cannot return, branch backwards, or throw exceptions. At splice time, JOIE replaces all **return** statements in the method splice with branches to the first statement immediately following the splice. All local frame accesses are redirected to a newly allocated segment of the frame.

While Mixin supports method splices only as prefixes or suffixes of existing methods, they are safe at the beginning or end of any basic block. The Core and Instrumentation interfaces support a wider range of method splices.

```
// loads methods and method splices from an auxiliary class
ClassInfo aux_class = new ClassInfo("examples.AuxClass");
Method new_method = aux_class.getMethod("newMethod");
Method pre_method = aux_class.getMethod("method_prefix");
Method post_method = aux_class.getMethod("method_suffix");

// adds a new method and field to the target class
target.addMethod(new_method);
target.addField(new Field(Type.INT, "counter"));

// brackets method splices from auxiliary class to a target method
Method targ_method = target.getMethod("run");
targ_method.addPrefix(pre_method);
targ_method.addSuffix(post_method);

// adds all fields, methods, and interfaces of the mixin to the target
ClassInfo mixin_class = new ClassInfo("examples.MixinImpl");
target.mixin(mixin_class);
```

Figure 6. Example use of Mixin Interface

```
Filter f = new InstructionFilter(Instruction.PUTFIELD);
Generator g = new Generator(aux_class.getMethod("splice1"));
Traversal t = Traversal.DOMINATING_PATH;
Instrument instr = new Instrument(f, g, t);
instr.instrument(method);
instr.instrument(cinfo);
```

Figure 7. Example use of Instrumentation Interface

```
public interface InstrumentTool {
    public void initClass(ClassInfo cinfo);
    public void endClass(ClassInfo cinfo);
    public void initMethod(Method method);
    public void endMethod(Method method);
}
```

Figure 8. InstrumentTool Interface

3.4. Instrumentation Interface

INSTRUMENTATION allows a transformer to splice new code at arbitrary points in existing code. INSTRUMENTATION is packaged as a generic, configurable `Instrument` class that a transformer may instantiate and apply to individual methods or to entire classes, as shown in Figure 7. `Instrument` may itself function as a complete transformer: it has `select` and `transform` methods, and may apply to an entire class.

Instrumentation behavior is driven by three control objects passed as arguments to the `Instrument` constructor. Each object implements a public JOIE-defined interface extending a common base interface called `InstrumentTool`, which defines methods to initialize the state of the tool when operating on new classes and methods (see Figure 8). `Instrument` calls these methods automatically before and after instrumenting classes and methods. The three `InstrumentTool` types are:

- A `Traversal` (see Figure 9) defines the order in which `Instrument` visits potential splice sites. Traversals support two key operations: advance to the next instruction, and advance to the next path. A *path* is a sequence of instructions whose meaning is traversal-dependent. JOIE includes predefined traversals to take a single linear path through each method, to treat each basic block as a path, or to perform a dominating path analysis, in which each path begins with the next basic block that is not guaranteed to be reached from some previous basic block. The more complex traversals enable transformers to minimize runtime overhead by including at most one splice per basic block or per dominating path. For example, the `Observable` transformer in Subsection 4.2 uses a dominating path traversal to minimize the number of splice sites for code that traps and records object updates.
- A `Filter` (see Figure 10) selects the insertion sites from among the instructions visited by a traversal. For example, the filter might insert a splice at every `putfield` instruction, every `putfield` to the local class, every static method invocation, etc. JOIE includes a predefined filter that selects on the bytecode form. For example, an instrumentation to trace all non-array allocation events at runtime would use a filter that selects all `new` instructions. Current filters examine one instruction at a time; regular expression pattern matching would be a useful extension.
- A `Generator` (see Figure 10) supplies the splice for each insertion site. The generator may statically define the splice, or it may generate the splice dynamically from the surrounding context. Figure 7 illustrates use of a predefined generator using a method splice, as described in Section 3.3. Although this specific generator and others are constrained, INSTRUMENTATION itself imposes no constraints on the spliced code produced by a generator.

A strong separation between these three building blocks improves reusability of transformer code, since a given generator, for example, can be used with any traversal or filter. Transformer authors can select from a library of predefined `InstrumentTool` classes, or define new ones as needed. `Instrument` coordinates the interactions among the components: it calls `next` and `nextPath` on the traversal, and passes each instruction to the filter. If the filter returns true, `Instrument` passes the traversal and the instruction to the generator.

```
public interface Traversal extends InstrumentTool {
    public boolean hasNext();
    public Instruction next();
    public boolean hasNextPath();
    public void nextPath();
    public void insertBefore(Splice s);
    public void insertAfter(Splice s);
}
```

Figure 9. `Traversal` Interface

```
public interface Filter extends InstrumentTool {
    public boolean filter(Instruction inst, Traversal t);
}

public interface Generator extends InstrumentTool {
    public Splice getSplice(Instruction inst, Traversal t);
}
```

Figure 10. `Filter` and `Generator` Interfaces

```
Instruction label = new Label();
Instruction i1 = new Branch(int flag, Label label);
Instruction i2 = new Constant(int ivalue);
Instruction i3 = new Getfield(Fieldref f, boolean is_static);
Instruction i4 = new Invoke(Methodref m, boolean is_static);
Instruction i5 = new Load(Type type, LocalVar local);
Instruction i6 = new TableSwitch(int[] values, Label[] targets);
```

Figure 11. Examples of SYMBOLIC Interface

Configuration methods control the behavior of the INSTRUMENTATION transformer. Depending on its configuration, the INSTRUMENTATION can instrument or ignore constructors, class initializers, static methods, or finalizers. Additionally, it can either instrument every instance that matches the filter, or just the first instance per path of the traversal. In the latter configuration, `Instrument` automatically short-circuits the remainder of any path in which `Filter` triggers a splice. Alternatively, the filter may perform the splice insertion itself, and return false to short-circuit the generator.

3.5. Symbolic Interface

Figure 11 illustrates the Symbolic interface, a flexible interface for creating and manipulating instructions. Symbolic may be used in concert with Instrumentation for on-the-fly splice generation or in other cases for which method splices are not sufficiently powerful. Like Bytestream, Symbolic represents instructions as instances of subclasses of the JOIE `Instruction` type; Symbolic adds new subclasses with methods appropriate for each instruction type. Symbolic insulates transformer authors from many bookkeeping details:

- Symbolic represents instruction operands and arguments as logical references to other objects rather than as numeric offsets into tables. For example, a `Branch` instruction instance contains a reference to the target `Instruction`, rather than a byte offset. Also, references to methods are represented by instances of the class `Methodref`, instead of as raw integers as in the Bytestream interface. A transformer can obtain a `Methodref` from the constant pool.
- Symbolic instruction classes contain logic to preserve referential integrity across changes to the classfile. For example, a `Branch` instruction automatically updates its offset field if new instructions appear between the branch and its target.
- Symbolic hides subtle distinctions among different forms of the same instruction. For example, the JVM specification defines fifty separate bytecode forms that can load or store a value, depending on the size and type of the value and its location in the frame. Symbolic presents unified `Load` and `Store` instruction classes that generate the correct bytecode form for the operands.

Users of Symbolic must understand the basic structure of the JVM architecture and the bytecodes, such as the existence of the frame, the stack-oriented nature of the instructions, and the constant pool.

Since Bytestream and Symbolic use a common `Instruction` type, either may be used with the other JOIE interfaces for manipulating lists of instructions (Core and Instrumentation). JOIE uses Symbolic for all instructions parsed from the classfile, and iterators from the Core interface return instances of Symbolic instructions.

3.6. Discussion

The Mixin, Instrumentation and Symbolic interfaces directly cover a wide range of common cases for bytecode transformers using code insertion. This insulates transformer authors from the complexity of the inner Core and Bytestream interfaces, and simplifies development of bytecode transformers that are safe as well as powerful. However, some tradeoffs are inherent in the layered architecture.

One drawback of a layered architecture in which the inner interfaces are exposed is that upper layers may maintain private state that is unknown to the inner layers. This raises the problem of keeping this state consistent when inner and outer interfaces are used in concert on the same target program. For example, Since Symbolic is layered above Bytestream, manipulations to instructions using Symbolic are reflected at the Bytestream level. However, the converse is not true: Symbolic supplements the intermediate representation with structures that are not updated by Bytestream. Thus transformers cannot safely use both Bytestream and Symbolic to manipulate the same instructions.

In general, it should not be necessary to use inner and outer interfaces in concert within a single transformer. The outer interfaces are specialized to meet common functional requirements without direct use of the inner interfaces.

A second issue is the need to check that transformer operations preserve the integrity of the target class. These include validity checks to protect against transformer errors, as well as consistency updates in the outer primitives to maintain invariants in the class representation. Examples of these consistency updates include updating the number of instructions in a method, reparsing the list of parameters after a change to a method signature, or ensuring that an entry is in the constant pool.

JOIE benefits from its layered architecture by leaving all integrity checks to the outer layers, to amortize their cost across batches of low-level updates in the outer primitives. However, it is difficult to safely batch integrity checks across sequences of outer primitives applied by the same transformer. For example, Symbolic invokes helper methods to ensure the representations are valid after every call. This improves safety, but it imposes a performance cost that can be avoided only by using Bytestream directly. If specific sequences of operations are common, then we can add new outer primitives to encapsulate them. This technique leverages the layered architecture to apply the integrity checks more efficiently.

4. Example Applications

This section presents three example applications of bytecode transformation using the JOIE code insertion interfaces: capability-based protection, observability, and object faulting. Each application uses a different style of code insertion: capability-based protection uses Mixin to extend the method interfaces of the target class, observability uses Instrumentation to modify the methods of the target class, and object faulting uses Symbolic and Instrumentation in concert to modify references to instances of a given class. These examples illustrate the nature and power of the code insertion interfaces described in the previous section. Together, they demonstrate the key building blocks for a range of rich applications of bytecode transformation using the JOIE interfaces.

4.1. Capability-Based Protection

The Java environment includes security mechanisms to protect the system from errors and malicious attacks, by preventing untrusted code modules from gaining unauthorized access to sensitive resources [8]. These sensitive resources include the local file system, the network, and runtime methods such as `exit`. Researchers have used bytecode transformation tools to extend the Java security system to make it stronger and more flexible, with the goal of enabling mutually distrusting code modules to coexist and work together in the same JVM [10][5][22][23]. Two of these systems, Safkasi [23] and Naccio [5], are based on JOIE.

```
public ClassInfo transform(ClassInfo cinfo) {
  ClassInfo splices = new ClassInfo("CapabilitySplices");

  Method cap_pre = splices.getMethod("prefix");
  cinfo.addPrefixes(cap_pre);

  Method cap_suf = splices.getMethod("suffix");
  cinfo.addSuffixes(cap_suf);

  ClassInfo mixin = new ClassInfo("CapabilityMixin");
  cinfo.mixin(mixin);

  return cinfo;
}
```

Figure 12. Implementation of Capability Transformer

```
public interface CapabilityProtected {
  // called by security manager to install a capability
  void initCapability(Capability c);

  // validates access for subsequent method invocation
  void presentCapability(Capability c);
}
```

Figure 13. Capability Interface

This section illustrates issues for security extensions by showing how to use bytecode insertion to implement basic support for *capability-based protection*. In this model, access to a protected resource requires the caller to first present an unforgeable token or *capability* demonstrating authority for the requested operation. This section shows a selective transformer (Section 2.2) that transforms the classes for protected resources to enforce capability protection. This example does not address transformation of the code that accesses protected classes, which is similar in principle to the object faulting example discussed in Section 4.3. Nor does it specify the policies used by an external security manager to construct and securely distribute capabilities; one option is to use capabilities simply to cache permissions from a traditional security manager.

Figure 12 shows the implementation of the transformer using the Mixin interface. A key function of the transformer is to extend the target class with an implementation of the interface `CapabilityProtected`, shown in Figure 13, by mixing in methods and fields of the auxiliary class `CapabilityMixin`. The `CapabilityProtected` interface allows a security manager to install capabilities for each object using `initCapability`, and it allows a caller to present a capability for validation using `presentCapability`. The `CapabilityMixin` class implements the methods and fields to support this interface. The second role of this MIXIN transformer is to add capability validation to the original methods of the target class, by bracketing each method with prefix and suffix splices drawn from the `CapabilitySplices` subclass of the auxiliary class `CapabilityMixin`. Since these splices are defined in a subclass of `CapabilityMixin`, they may access its fields, e.g., the valid capabilities.

This simple example illustrates the power of MIXIN to extend target classes for capability protection or similar features. The implementation of the new feature in the auxiliary class `CapabilityMixin` is generic because it is orthogonal to the original function of the target class. The selective transformer provides a way to add this generic functionality to existing classes without adding boilerplate code by hand, as an alternative to restructuring the class hierarchy so that all protected resource classes inherit from a common superclass. The use of a prefix and suffix is a key ingredient that enables an automatic "wrapping" of the target class: the prefix/suffix wrapper interposes new behavior on the original methods, while the new methods control the behavior of the wrapper. In this respect, transformation using MIXIN is more powerful than inheritance as a mechanism to extend classes.

Note that the transformer itself is also generic; it contains none of the body of the code for handling capabilities. In the simplest case, the prefix validates that the client has used `presentCapability` to present a valid capability, while the suffix updates internal state to determine when a subsequent method invocation requires a new call to `presentCapability`. A more sophisticated implementation of `CapabilityMixin` could support richer policies for handling capabilities, such as multiple capabilities with selective revocation, or capability-based protection of specific methods. MIXIN enables this decoupling of application logic from transformation logic, improving modularity and insulating the author of the new feature from much of the complexity of bytecode transformation.

This example also illustrates a key limitation of the MIXIN interface. Since capability protection for the class is based on new code in the class methods, this approach cannot ensure capability protection for direct accesses to public fields in the class. One alternative is to transform classes that contain code that accesses objects of the protected class, instead of or in addition to transforming the protected class itself. Such a transformation requires use of INSTRUMENTATION, and is similar to the example discussed in Section 4.3. A complete capability system needs a transformer of this style to make capability handling transparent at access sites for protected objects, and to ensure that calls to `initCapability` originate only from a trusted security manager.

4.2. Observable

An *observable* object exports an interface for other objects to subscribe to it as *observers*. An observable object notifies its observers whenever its state changes. Observer/observable is a common design pattern [7], also called *model/view* and *publish/subscribe*. Examples of applications employing this design pattern include visual displays, debuggers, and write capture for object caching, persistence, and replication.

The standard Java class library includes an implementation of observer/observable. This implementation requires that observable classes subclass `java.util.Observable` to obtain

```
1  public void transform(ClassInfo cinfo) {
2    ClassInfo splices = new ClassInfo("ObservableSplices");
3    Method notify = splices.getMethod("notify_splice");
4    cinfo.addSuffixes(notify);
5    Filter filter = new InstructionFilter(Instruction.PUTFIELD);
6    Method setDirty = splices.getMethod("setDirty_splice");
7    Generator gtor = new MethodSpliceGenerator(setDirty);
8    Instrument instr = new Instrument(filter, gtor, Traversal.Dominator);
9    instr.setSemantics(Instrument.ONCE_PER_PATH);
10   cinfo = instr.transform(cinfo);
11   ClassInfo impl = new ClassInfo("ObservableImpl");
12   cinfo.mixin(impl);
13 }
```

Figure 14. Implementation of Observable Transformer

```
14 class LocalWrite implements Filter {
15   public boolean filter(Instruction inst) {
16     return (inst.form == Instruction.PUTFIELD &&
17         inst.stackArgument(0) == Instruction.THIS);
18   }
19 }
```

Figure 15. Implementation of Observable Filter

methods to register and notify observers. It relies on the programmer of the observable class to manually add code to notify its registered observers of internal state changes.

This section outlines a transformer that automatically adds the interfaces and internal support for an observable class. It does not require subclassing, and it automatically inserts code into the original methods to mark a modified object as "dirty" and notify its observers. These changes entirely lift from the programmer the burden of making an object observable. We used a similar approach to build Ivory, a distributed replication system [2].

This example demonstrates use of JOIE's code insertion interfaces to modify the body of existing methods. As shown in the previous section, MIXIN can extend the interface of the target class and add prefixes or suffixes to wrap existing methods. However, it is not sufficiently powerful to modify the original code of the methods, for example, to detect modifications to the object's internal state as required for observable. The observable transformer illustrates use of MIXIN in concert with INSTRUMENTATION to detect and respond to object updates made by the target class methods.

Figure 14 shows the implementation of the observable transformer. Lines 2-4 use MIXIN to add a suffix to the existing methods in a manner similar to the capability example. The `notify_splice` suffix, drawn from the auxiliary class `ObservableSplices`, inspects a dirty bit

```
public ClassInfo transform(ClassInfo cinfo) {

  // transform faultable access sites
  ProxyFilter cf = new ProxyFilter();
  ProxySplice splice = new ProxySplice();
  cinfo.instrument(cf, splice, Traversal.All);

  return cinfo;
}
```

Figure 16. Implementation of Object Faulting Transformer

to determine if the method modified the object state, and notifies observers accordingly. The remainder of the transformer deals primarily with adding and setting the dirty bit.

Lines 5-10 add new code to the body of each method to set the dirty bit if the object is modified. This sequence uses INSTRUMENTATION to insert a method splice `setDirty_splice`, also drawn from `ObservableSplices`. Figure 15 shows the filter instantiated at line 5 of Figure 14. It selects instrumentation sites at instances of the `putfield` instruction, the JVM bytecode responsible for writing a new value to an object field. It also checks that the first stack value consumed by the `putfield` instruction is placed on the stack by the instruction `aload_0`; this verifies that the reference is to the current object. The generator injects `setDirty_splice` to mark the object as dirty after the `putfield` instruction.

The instrumentation shown in lines 5-10 of Figure 14 uses `Traversal.Dominator`, a predefined dominating path traversal described in Section 3.4. `ONCE_PER_PATH` semantics specifies that the traversal skips all instructions dominated by the current block after each splice. These traversal attributes ensure that the dirty bit is set when necessary in the most efficient way, by skipping exactly those instructions guaranteed to execute only after the dirty bit has already been set. If instead the transformer used `Traversal.All` it would instrument every `putfield` in each method; if it used `Traversal.Blocks` with `ONCE_PER_PATH` it would instrument the first `putfield` of each basic block.

Finally, lines 11-12 mix in the auxiliary class in `ObservableImpl`, which defines the dirty bit field and the subscription and notification methods for interacting with observers. This mixin also marks the class as implementing the interface `joie.examples.Observable`, which does not require inheritance as `java.util.Observable` does.

Like the capability transformer in Section 4.1, the observable transformer adds a feature to selected target classes by modifying only those classes. This example shares a limitation of the capability transformer example: it does not examine or transform accesses to an observable object appearing in other classes. Given this limitation, the observable transformer cannot detect object writes other than those resulting from a local `putfield`, e.g., it cannot detect external writes to public fields of an observable object. Doing so requires a more comprehensive whole-program transformation. The following section presents an example of such a transformation.

4.3. Object Faulting

This section presents a whole-program transformation for *object faulting.* A *proxy* object stands in for an object residing in external storage or at a remote site. In this example, a local access to a proxy triggers an *object fault* that locates the object and instantiates it locally, transparently replacing the proxy. This object faulting example is a powerful building block for systems that cache persistent or distributed data, or that generate data structures on demand. The approach is similar to pointer swizzling at page-fault time[25] for persistent systems, in that a faulted object may in turn contain other external references that are resolved lazily in a "wavefront" fashion as local threads traverse the data structure.

This application uses two transformers to enable applications to use object faulting. The first, a selective transformation similar to the earlier two examples, transforms target classes to enable them to be proxied and resolved. It does so by marking target classes as implementing the interface `FaultableObject`, adding a boolean variable `here` to indicate whether each instance is resident, and adding a new method, `resolve`, which locates a nonresident object and instantiates it locally.

The second transformer is a whole-program transformation designed to allow code written with no knowledge of faulting proxies to access them transparently. The transformer splices object faulting code to supplement each potential access to a faultable object with a test to trap accesses to nonresident (proxy) objects before they occur. If the referenced object is not resident, the splice invokes a `resolve` method to fetch the object before the access executes. In this system, the object and its proxy are instances of the same class, and the system resolves a fault by overwriting the proxy in place with the object. This avoids the need to update ("snap" or "swizzle") references to the proxy in other objects. The holder of a reference need not care if the reference points to a resident object or its proxy.

Figure 16 shows the object faulting transformer. The transformer is similar to the previous examples in its use of INSTRUMENTATION, but it uses SYMBOLIC to generate a custom splice for each access site. As a whole-program transformation, it applies to any class referencing a faultable object. The transformer edits all references to instances of `FaultableObject`, inserting code to check residency and resolve nonresident objects before any access.

This example is a good illustration of code transformation for a stack machine. Before the target bytecode can access any object, it must first push a reference to that object onto the stack. The transformer actually injects residency checks at these *push sites* rather than at the access sites, because it can determine more information about the target of the reference at the push site. Only six operations can push an object reference onto the stack: new allocation, field dereference, array dereference, copy from the frame (i.e., a local variable or a method parameter), duplicate, and return from a method invocation. Newly created objects are by definition always resident. Object references that are copied from the frame, duplicated, or returned from a method must first be pushed on the stack by one of the other operations. Thus it is sufficient to instrument just the push sites that obtain a stored reference from an object field or from an array element. These accesses use one of three bytecodes: `getfield` fetches a non-static field, `getstatic` fetches a static field, and `aaload` fetches from an array element.

```
1 class ProxyFilter implements Filter {
2   public boolean filter(Instruction inst) {
3       if(inst.form == Instruction.GETFIELD ||
4               inst.form == Instruction.GETSTATIC) {
5           return faultable_xformer.select(inst.getType());
6       }

7       if(inst.form == Instruction.AALOAD) {
8           return true;
9       }

10       return false;
11   }
}
```

Figure 17. Implementation of Proxy Filter

Lines 4-6 in Figure 16 use Instrumentation in concert with Symbolic to identify these push sites and insert the residency checks. The instrumentation is more sophisticated than the previous examples in that it uses a custom code generator (`ProxySplice`) for the splice instead of inserting a static method, and it uses a more discriminating filter. Also, it uses `Traversal.All` to traverse all instructions of each method to ensure that it transforms all qualifying push sites.

Figure 17 shows the implementation of the filter to select the push sites where faults may occur. Line 5 tests to determine if the value retrieved by a `getfield` or a `getstatic` is a reference to a `FaultableObject`. It does this by using the `select` method (see Section 2.2 of the selective transformer to see if that class has been (or will be) transformed. The transformer cannot easily make this check statically for values retrieved by `aaload`, because `aaload` does not encode the type of the object, as `getfield` and `getstatic` do. Thus the filter conservatively selects every `aaload` site for instrumentation; the splice for these sites checks the target type at runtime, as described below.

The transformer uses Symbolic to generate a custom splice for each push site selected by the filter. The transformer must use a custom splice for three reasons. First, the required bytecode sequence cannot be compiled from Java language source. Second, the splice references a `FaultableObject` field and method through the constant pool of the transformed class; the representation of this reference may be different for every class. Finally, the array reference case (`aaload`) requires an additional runtime type check. Although the `ProxyFilter` can distinguish this case from the `getfield` and `getstatic` cases, Instrumentation as defined is not powerful enough to select a specific generator for each case.

Figure 18 shows an example of the generated splice. The splice consists of three segments. The first segment is a runtime type check to determine if the target of the pushed reference is an instance of `FaultableObject`. This segment only appears at push sites using array references (`aaload`), since the transformer executes this check statically in `ProxyFilter` for `getfield`

```
0  // original getfield|getstatic|aaload instruction
1  DUP                         // only for aaload
2  INSTANCEOF FaultableObject      // only for aaload
3  IFEQ 9                      // only for aaload

4  DUP
5  GETFIELD <FaultableObject.here>
6  IFNE 9

7  DUP
8  INVOKESTATIC ProxyRegistry.resolve(FaultableObject)
9  // original instruction following getfield|getstatic|aaload
```

Figure 18. Bytecodes of Proxy Splice

and `getstatic`. The second segment checks the `FaultableObject` resident flag `here`. If the object is not resident, the third segment triggers a fault by invoking a `resolve` method to fetch the object. Each segment consumes the object reference from the stack; the splice keeps each segment stack-neutral by prefixing it with a `DUP` instruction, so the splice leaves the reference on top of the stack when it completes.

Figure 19 shows the generator for the custom splice. The generator code embodies a generic template for the splice, with an explicit `Label` instruction that acts as a branch target to jump over the second and/or third splice segments if the object is not faultable or is already resident. For these checks, the splice references the `FaultableObject` class and its `here` field through the constant pool of the transformed class. Lines 3-10 obtain these references from the constant pool as the transformer initializes for each target class; Symbolic primitives automatically generate the correct bytecodes for these references. Finally, line 27 contains the method `endMethod`, allowing the splice to increase the method's maximum stack.

4.4. Splices in Concert

These examples are meant to be illustrative, and to suggest the range of possible applications of the outer Mixin, Instrumentation, and Symbolic interfaces for code insertion in JOIE. In particular, they illustrate three important mechanisms of code insertion: interface extension, method modification, and call site modification.

Using these code insertion mechanisms in concert enables a wide range of powerful transformations. For example, interface extension and call-site modification can work together to transform a target application so that a system can securely record and act on the application's state. The transformer extends the interface of each target class with new methods or parameters, while adjusting reference sites to use the new interfaces instead of or in addition to the old ones. At the same time, it verifies that the original (untransformed) code does reference the extended interfaces.

```
class ProxySplice implements Splice {
   Reference here, resolve, proxyobj;

   public void initClass(ClassInfo cinfo) {
       ConstantPool cpool = cinfo.getConstantPool();
       here = cpool.getFieldRef("FaultableObject", "here", "Z");
       resolve = cpool.getMethodRef("FaultableObject", "resolve",
           "(LFaultableObject;)V");
       proxyobj = cpool.getClassRef("FaultableObject");
   }

    public void generator(Instruction inst) {
       Splice frag = new Splice();
       Label label = new Label();

       if(inst.form == Instruction.AALOAD) {
           frag.add(new Dup());
           frag.add(new InstanceOf(proxyobj));
           frag.add(new Branch(Branch.IF_FALSE, label));
       }

       frag.add(new Dup());
       frag.add(new Getfield(here));
       frag.add(new Branch(Branch.IF_TRUE, label));

       frag.add(new Dup());
       frag.add(new Invoke(resolve));
       frag.add(label);

       return frag;
   }

   public int endMethod(Method meth) {meth.getCode().addStack(1);}}
```

Figure 19. Implementation of Proxy Splice

This technique is a powerful building block for adding security or resource management features to Java-based systems. In the capabilities example (Section 4.1), the system cannot trust client programs to access only those objects for which they hold capabilities, or to release them when they are done. If a matching whole-program transformation inserts all calls to request, present, and release the capabilities, the security system may trust that the application cannot use those interfaces maliciously. This approach is used in Safkasi [23], which adds a new parameter encapsulating the current security context at call sites and at receiving methods. This parameter is secure because it can be manipulated only by the inserted code. The authors of Safkasi refer to this hidden passing convention as *security-style passing*.

5. Conclusion

This paper explores support for automatic code insertion in the JOIE bytecode transformation architecture. JOIE is a toolkit for creating Java *transformer* programs that modify compiled application code packaged as Java class files. Researchers have used JOIE transformers in systems for security, resource management, debugging, and data replication and persistence. We use the JOIE transformation architecture to illustrate key issues for code transformation in the Java environment.

A key design goal of the JOIE architecture is to balance conflicting requirements of safety, ease of use, performance, and expressiveness of the interface. JOIE meets these goals through a layered architecture incorporating specialized "outer" interfaces to simplify common styles of code insertion, layered above unrestricted low-level interfaces for accessing and editing classfiles. We illustrate the strengths and limitations of this architecture by presenting three high-level interfaces for code insertion: SYMBOLIC to modify or synthesize instruction sequences, MIXIN to extend target classes by injecting attributes from auxiliary classes written in Java, and INSTRUMENTATION to select sites for code insertion using modular traversal and filtering primitives. These outer interfaces are expressive, safe, and easy to use.

We present three example applications — capability-based security, observer/observable, and object faulting — to demonstrate the range of uses of bytecode transformers and their implementation with the JOIE code insertion interfaces. These examples demonstrate that bytecode transformation is a powerful and useful technique for extending Java programs by injecting new features after compilation. The transformation approach enables a decoupling of independent concerns (e.g., system-level features such as security or data management) from core applications, simplifying development and encouraging code reuse. The new features are decoupled from the application programs and implemented once — in the transformer — rather than reimplemented in each program.

JOIE, its source code, and the applications described in this paper are publically accessible for research and experimentation. They can be found at `http://www.cs.duke.edu/ari/joie/`.

REFERENCES

1. O. Agesin, S. Freund, and J.C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the Symposium on Object Oriented Programming: Systems, Languages, and Applications*, pages 49–65, 1997.
2. Geoff C. Berry, Jeffrey S. Chase, Geoff A. Cohen, Landon P. Cox, and Amin Vahdat. Toward Automatic State Management for Dynamic Web Services. In *Proceedings of the Network Storage Symposium*, October 1999.
3. Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic Program Transformation with JOIE. In *USENIX 1998 Annual Technical Conference*, pages 167–178, June 1998.
4. Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Conference Proceedings of the 1998 ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, October 1998.
5. David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *1999 IEEE Symposium on Security and Privacy*, May 1999.
6. Michael S. Fulkerson and Alan W. Biermann. Javox: A Toolkit for Building Speech-Enabled Applications. In *The 2000 Applied Natural Language Processing Conference*, May 2000.

7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, Reading, Massachusetts, 1995.
8. Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New security Architecture in the Java Development Kit 1.2. In *Proceedings of the USITS Symposium on Internet Technologies and Systems*, December 1997.
9. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley Publishing Company, Reading, Massachusetts, 1996.
10. Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *1998 USENIX Annual Technical Conference*, June 1998.
11. Ralph Keller and Urs Hölzle. Binary Component Adaptation. Technical Report TRCS97-20, Department of Computer Science, University of California at Santa Barbara, December 1997.
12. Gregor Kiczales. AspectJ: How to Use Aspect-Oriented Programming to Solve Common Modularity Problems in Java Programs. In *PARCForum*, 2000.
13. Butler Lampson. Hints for Computer System Design. *Operating Systems Review*, 15:33–48, October 1983.
14. James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 1995*, June 1995.
15. Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.
16. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley Publishing Company, Reading, Massachusetts, 1997.
17. Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the 1997 USENIX NT Conference*, 1997.
18. Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–216, December 1999.
19. Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.
20. Daniel Vogelheim. *Ein Rahmenwerk zur Untersuchung zustandsbasierter Testverfahren fr objektorientierte Software.* Lehr- und Forschungsgebiet Informatik III. RWTH Aachen, 1999.
21. D. W. Wall. Systems for late code modification. In Robert Giegerich and Susan L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.
22. Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 116–128, October 1997.
23. Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *1998 IEEE Symposium on Security and Privacy*, May 1998.
24. Ian Welch and Robert Stroud. *Dalang* – a reflective extension for Java. Technical Report CS-TR-672, Department of Computer Science, University of Newcastle-upon-Tyne, September 1999.
25. Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *1992 International Workshop on Object Orientation in Operating Systems*, pages 364–377, 1992.