Regular Paper

A Case Study of Development of a Java Bytecode Analyzer Framework Using AspectJ

Susumu Yamazaki,[†1,†2] Michihiro Matsumoto,[†1,†2]
Tsuneo Nakanishi,[†2,†3,†4] Teruaki Kitasuka[†2]
and Akira Fukuda[†2,†3]

Aspect-orientation is a new programming paradigm that can localize a cross-cutting concern in a single module. This paper proposes a new type of Java bytecode analyzer framework based on aspect-orientation. It includes several new design and implementation techniques that are general or specific to the domain of language systems. We also observe that aspect-orientation improves extensibility, type safety, execution efficiency, and simplicity of the API, when compared with existing analyzer frameworks based on object-orientation such as Soot. This paper reports the following: structural extension of elementary objects maintaining type safety and execution efficiency; separation of a bytecode parser and concrete instruction sets; a visitor based on the stack-machine model; binary operations that are simple, extensive, and easy to maintain; and separation of nonfunctional concerns such as verification. We also observe that AspectJ currently has two limitations: it is not sufficiently expressive to structure aspects strongly depending on the inner structure; and it does not provide a general approach to write advice that cannot be described with information of its pointcut only.

1. Introduction

A Java bytecode analyzer framework has a wide range of applications including bytecode-level optimizing compilers, ahead-of-time compilers, verifiers, and aspect weavers. It will form the basis of software development supporting tools that realize a development methodology that we are developing.

Soot [20], a popular framework, was created based on object-orientation. Its characteristic design policy is its capability of simple extension. Consequently, it experiences some problems in separation of concerns, type safety, execution efficiency, complexity of API, and others.

This problem can be solved using Aspect-oriented programming (AOP)[15], which is a new programming paradigm for separation of concerns. AOP can localize and modularize a cross-cutting concern that conventional (object-oriented, procedural) programming paradigms cannot localize, or modularize, to a single place in the system. For example, a logging process that records a method enter and exit cannot be realized with object-orientation unless we insert that process into all methods in the system. On the contrary, we can easily add the logging process to the system with aspect-orientation, to describe a logging aspect, which is a kind of module.

This paper proposes a new type of Java bytecode analyzer framework based on aspect-orientation using AspectJ [14]. We also observe that aspect-orientation improves the separation of concerns, extensibility, type safety, execution efficiency, and simplicity of API, when compared with existing analyzer frameworks based on object-orientation.

Our contributions are the following:

- We propose four techniques on AspectJ:
  (1) Direct Extension
  (2) Abstract Category
  (3) Smart Visitor
  (4) Binary Operation
- We assess the effectiveness of these techniques.
- We also observe limitations in the current AspectJ when we describe a verifier as an aspect.

The rest of this paper is organized as follows. Section 2 describes an overview of our bytecode analyzer framework. Section 3 proposes the

†1 Fukuoka Laboratory for Emerging & Enabling Technology of SoC, Fukuoka Industry, Science & Technology Foundation
†2 Graduate School of Information Science and Electrical Engineering, Kyushu University
†3 Computing and Communications Center, Kyushu University
†4 System LSI Research Center, Kyushu University

This paper is an extended version of Ref. 23).

direct extension technique, which is the structured extension of elementary objects maintaining type safety and execution efficiency. Section 4 proposes the abstract category technique, which is the separation of an extendable bytecode parser from instruction sets. Section 5 proposes the smart visitor technique, which is a simple process description of each instruction. Section 6 proposes the binary operation technique, which is simpler and more extendable than double-dispatch. Section 7 discusses problems of description of a verifier as an aspect. Section 8 shows assessment of these techniques and discusses the result. Section 9 discusses some related works. Section 10 concludes this paper.

2. Framework Overview

Our framework includes the following parts:

- **Bytecode scanner**: iterates an instruction sequence of the bytecode in the class file. This uses Javassist [2] as a bytecode reader.
- **Bytecode parser**: translates the instruction sequence into *a code container*, which contains some *basic blocks*; it contains some *instruction objects*. This parser builds an intraprocedural factored control flow graph [4].
- **Instruction visitor**: receives an instruction object and dispatches the process corresponding to the instruction.
- **Data-flow analyzer**: analyzes data-flow information based on the Monotone framework [17]. This supports intra-/interprocedural analysis. *Type checker*, which determines types of variables in the program code, is one of its applications.

3. Direct Extension: Extensions to Elementary Objects

It is often remarked that aspect-oriented programming improves separation of concerns. We point out that the most effective example of this fact is that of elementary objects of a framework, such as instruction objects.

For example, consider adding a new feature to an analyzer derived from a framework: a simple approach is to add fields or methods to elementary objects to store necessary information.

However, traditional object-oriented programming languages cannot add fields or methods to elementary objects structurally. For that reason, they tend to 'bloat' chaotically. The class hierarchy can become deep if a structure is enforced. In either case, maintainability and readability are degraded.

Within a framework, the extension of elementary objects is realized using an indirect approach, such as table or the Visitor Pattern [7], rather than by direct addition of fields or methods. For example, elementary objects are extended by adding tags in Soot. Tags are named. They may be requested by searching a table using this name.

The above techniques may sacrifice type safety or execution efficiency. Soot sacrifices both of them: the retrieved tag sacrifices type safety and must be cast downward before it may be used. Execution is inefficient because of the need for searching the table.

AspectJ can define fields and methods directly using classifications as an aspect using an inter-type member declaration. For example, if we add a field or a method necessary for an analysis, we can define it structurally in an aspect concerned with the analysis.

Indirect extension mechanisms, such as tags in Soot, are no longer needed. The type system of AspectJ ensures the type safety of added fields and methods. Execution efficiency is improved when compared to indirect extension because they are woven into classes directly.

4. Abstract Category: Separation of the Bytecode Parser and Instruction Sets

A bytecode parser scans binary class files, generates instruction objects corresponding to the byte sequences, and inserts labels. It also sets the relationships between instructions. It does so, for example, by using a succeed set, which is a set of instructions that may be executed after other instructions, except those throwing exceptions, in a manner similar to that of a Factored Control Flow Graph [4].

Next, we specifically address setting succeed sets, which depend on the class of an instruction. Instructions are divided into non-terminator and terminator categories: a succeed set of a non-terminator includes the next instruction, while a succeed set of a terminator does not.

Instructions may also be divided into non-branch, branch, and switch categories: a succeed set of a non-branch instruction does not include any special jump target; a succeed set of a branch instruction includes one jump tar-

get; and a succeed set of a switch instruction includes two or more jump targets.

A succeed set can be determined by classification, rather than by the instruction set, but the instruction set determines how a concrete instruction class is classified. In addition, another process, such as one detecting potential exception-throwing instructions (PEIs)[4], may require another classification. Therefore, we must realize such a classification using `interface` because Java is a language that supports single inheritance and multiple supertypes.

However, Java does not allow `interface` to have concrete methods. Therefore, the process of setting succeed sets is distributed among code sections that contain concrete instructions.

AspectJ solves this problem. First, we provide two aspects to the parser. The first aspect is `addNextToSucc`, which adds the next instruction to the succeed set if the current instruction is a non-terminator. The second aspect is `addBranchTargetToSucc`, which adds the target instruction(s) to the succeed set if the current instruction is a branch or a switch. Secondly, we make a concrete instruction class implement the `interface` corresponding to the classification. Lastly, if the order of the succeed set is important, we can set the priority order using the `precedence` declaration between `addNextToSucc` and `addBranchTargetToSucc`. **Figures 1** and **2** show example codes of a parser and an instruction set.

Generally speaking, if there are classifications into some given classes, and if the classifications determine the corresponding processes, we can write simple code so that the classifications and the processes are represented using `interface` and aspects, respectively.

Although multiple inheritance has similar effects, this approach using aspects has two advantages: we can add a classification without modifying existing code. In addition, we can avoid the method conflict problem. For example, an instruction that is both a non-terminator and a branch is realized easily in AspectJ but cannot be realized naturally using multiple inheritance.

5. Smart Visitor: The Specialized Visitor for the Stack Machine Model

An operation corresponding to a given instruction often includes common processes. For example, because Java bytecode is based on the stack machine model, operations such as push or pop are commonly included in the operations corresponding to each instruction.

Therefore, we provide a Smart Instruction Visitor as part of our framework, based on the Java bytecode model, which is a domain-

```
public class Parser {
    public static class Instruction {
        void setSucc
          (Instruction[] table, int pc) {}
        ...
    }
    public static interface Terminator {}
    public static interface Branch {...}
    public static interface Switch {...}
    static aspect addNextToSucc {
        pointcut addNextToSucc(Instruction inst,
          Instruction[] table, int pc)
            : call(void Instruction.setSucc
                   (Instruction[], int))
            && target(inst) && args(table, pc)
            && !target(Terminator);

        before(Instruction inst,
            Instruction[] table, int pc)
            : addNextToSucc(inst, table, pc) {
            ...
        }
    }
    static aspect addBranchTargetToSucc {
        pointcut addBranchTargetToSucc
          (Instruction inst,
           Instruction[] table, int pc)
            : call(void Instruction.setSucc
                   (Instruction[], int))
            && target(inst) && args(table, pc)
            && target(Branch);

        before(Instruction inst,
         Instruction[] table, int pc)
            : addBranchTargetToSucc(inst,
                 table, pc) {
            ...
        }
        ...
    }
}
```

Fig. 1 Bytecode parser using AspectJ.

```
import Parser.*;

public class Aload extends Instruction {...}
public class Return extends Instruction
    implements Terminator {...}
public class Ifeq extends Instruction
    implements Branch {...}
public class Goto extends Instruction
    implements Terminator, Branch {...}
public class Tableswitch extends Instruction
    implements Terminator, Switch {...}
...
```

Fig. 2 Java bytecode instruction set example.

```
public abstract class InstructionVisitor {
    ... // S1
    protected void push(Object value) {}
    protected void push2(Object value) {
        push(value);
    }
    ...
    protected void pushInt(Object value) {
        push(value);
    }
    ...
    protected void pushDouble(Object value) {
        push2(value);
    }
    ...
    protected Object pop() {
        return null;
    }
    ...
    protected void store
      (int index, Object value) {}
    ...
    protected Object load(int index) {
        return null;
    }
    ... // S2
    public static abstract aspect Pointcuts {
        pointcut intBinaryOperator
          (InstructionVisitor v,
           Instruction inst,
           Object value1, Object value2)
           : execution
             (Object InstructionVisitor+
              .at(Instruction+, Object, Object))
             && target(v)
             && args(inst, value1, value2)
             && args(Idiv, Object, Object)
             && ...;
        ...
    } // S3
    protected Object at
      (Iload inst, Object loadedValue) {
        return loadedValue;
    }
    protected Object at
      (Idiv inst,
       Object value1, Object value2) {
        return null;
    }
    ... // S4
    static aspect InsertCode {
        private abstract void Instruction.at
          (InstructionVisitor v);
        ...
        private void Iload.at
          (InstructionVisitor v) {
            Object value
              = v.loadInt(this.index);
            value = v.at(this, value);
            v.pushInt(value);
        }
        private void Idiv.at
          (InstructionVisitor v) {
            Object value2 = v.popInt();
            Object value1 = v.popInt();
            Object result
              = v.at(this, value1, value2);
            v.pushInt(result);
        }
        ...
    }
}
```

Fig. 3 The smart instruction visitor.

specific variant of the Visitor Pattern [7]. The programmer has access to four basic operations (`push`, `pop`, `load`, `store`) and the processes corresponding to each instruction. The programmer does not need to write all of these and can override only those necessary.

We provide variations of basic operations corresponding to different types because Java bytecode is a typed language. For example, `pushInt` corresponds to the type `int`. We also provide `push` and `pop` operations that handle values using appropriate types for `getfield`, `putstatic`, etc. Moreover, we provide variations for basic operations corresponding to 32- and 64-bit types to satisfy the Java bytecode specification. Finally, we provide variations of `push` and `pop` corresponding to either two 32-bit types or one 64-bit type, for `dup2`, etc.

Our framework describes the process corresponding to each instruction as a method which is supplied an instruction object and zero or more arguments, and which returns a zero or one result. For example, a process corresponding to the instruction `idiv` is defined as a method that is given an instruction object and two division values; moreover, it returns a result value object.

It is effective to define pointcuts for methods corresponding to instructions that have common features. This allows the methods to be defined structurally from various viewpoints.

Figure 3 shows an implementation of the Smart Instruction Visitor. Basic operations, various pointcuts, processes corresponding to instructions and inner processes, are defined from `S1`, `S2`, `S3`, and `S4`, respectively.

The basic behavior is as follows: Methods receiving a Visitor are first defined using intertype method declarations (`S4`). The corresponding basic operations and processes are called in these methods. For example, the inner method of `idiv` calls `popInt`, twice. The process corresponding to `idiv` is called with the instruction object and the returned values; `pushInt` is called with the returned value.

We provide default implementations of basic operations and processes that correspond to each instruction. Relationships between variations of basic operations are represented as an invocation from more constrained variation methods to less constrained (`S1`). Therefore, all a programmer must do is to override the necessary methods.

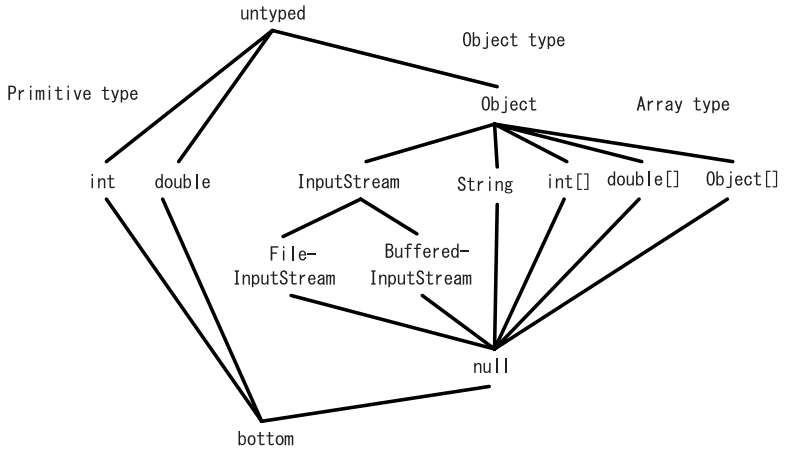

Fig. 4 The type property space for java.

6. Binary Operation: Simple and Extensible Implementation

We must implement the binary operation of properties to realize a data flow equation as a framework. In type checking, for example, we must calculate the least upper bound ($\cup$) of the type property at the merge points [17].

Figure 4 shows a lattice representing the property space for type checking [16]. Bottom $\bot$ represents an initial value. Therefore, the least upper bound of property P and $\bot$ is P ($\bot \cup P = P \cup \bot = P$). Top $\top$ in type checking indicates untyped. The least upper bound of property P and $\top$ is $\top$ ($\top \cup P = P \cup \top = \top$).

Next, the least upper bound of the same primitive type, such as `int`, is the type and the least upper bound of a different primitive type is untyped. For example, $P_{\texttt{int}} \cup P_{\texttt{int}} = P_{\texttt{int}}, P_{\texttt{int}} \cup P_{\texttt{float}} = \top$. Note that the rule of top and bottom precedes this rule, *i.e.* $P_{\texttt{int}} \cup \bot = \bot$.

Next, the least upper bound of the object type is a common ancestor. For example, $P_{\texttt{FileInputStream}} \cup P_{\texttt{BufferedInputStream}} = P_{\texttt{InputStream}}$. Note that the rules for bottom, top, and primitive types precede this rule. Moreover, the least upper bound of an object type is a class with zero or more interfaces because Java provides single inheritance of class, but multiple subtyping of interfaces.

Finally, the least upper bound of `null` and the primitive type are untyped; the least upper bound of `null` and the object type are the object type. Note that the rule of bottom precedes this rule.

```
public class PrimitiveType extends Property {
    public Property meet(Property p) {
        if(p instanceof Bottom) {
            return this;
        }
        if(p instanceof Untyped) {
            return p;
        }
        if(p instanceof ...) ...
        ...
    }
}
```

Fig. 5 Implementation of the primitive type property using `instanceof`.

Consider the implementation of binary operations with a least upper bound based on these rules. A naive implementation may use `instanceof`. For example, **Fig. 5** shows the implementation of a primitive type property, but this implementation is less extensible and maintainable. If we add a new type property, we must modify all `meet` methods, which calculate the least upper bounds. Moreover, if we change the order of precedence of the rules, we must swap the order of `if` in some methods.

Next, consider implementation using double-dispatch. **Figure 6** shows a binary operation using double-dispatch. The behavior of this operation is somewhat complicated. When the method `meet` is called, it calls the method `with*` corresponding to the class of the receiver `this`. For example, if the receiver is `Bottom`, it calls the method `withBottom`. The receiver and the argument of the call are swapped. This realizes binary operations by defining processes that correspond to each class of receiver and the argument of `meet`.

The advantage of double-dispatch is that

```
public abstract class Property {
    public abstract
      Property meet(Property p);
    protected abstract
      Property withBottom(Bottom p);
    protected abstract
      Property withUntyped(Untyped p);
    protected abstract
      Property withPrimitiveType
        (PrimitiveType p);
    ...
}
public class Bottom extends Property {
    public Property meet(Property p) {
        p.withBottom(this);
    }
    public Property withBottom(Bottom p) {
        return p;
    }
    public Property withUntyped(Untyped p) {
        return p;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        return p;
    }
    ...
}
public class Untyped extends Property {
    public Property meet(Property p) {
        p.withUntyped(this);
    }
    public Property withBottom(Bottom p) {
        return this;
    }
    public Property withUntyped(Untyped p) {
        return this;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        return this;
    }
    ...
}
public class PrimitiveType
  extends Property {
    public Property meet(Property p) {
        p.withPrimitiveType(this);
    }
    public Property withBottom(Bottom p) {
        return this;
    }
    public Property withUntyped(Untyped p) {
        return p;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        ...
    }
    ...
}
```

Fig. 6 Implementation of the type property using double-dispatch.

maintainability is improved because each method is simplified. However, the problems remain, as we must modify all property classes to add a new type property. We also must modify many classes to swap the precedence order of the rules. Moreover, we need to write the same process as many methods, such as the implementation of `Bottom` and `Untyped`.

AspectJ solves these problems simply (see **Fig. 7**). Actual processes are implemented using `around` without `proceed` in the coordinator aspect. For example, `BottomCoordinator` describes the process of involving the bottom and something else. Moreover, the process of combining different types is described in a *combination coordinator*. For example, a process of involving a combination of object types and primitive types is described in `ObjectAndPrimitiveCoordinator`.

Next, coordinators are sorted in `precedence` in topological order of lattice from the bottom. A combination coordinator precedes the coordinator of each property. The content of the method `meet` in the class `Property` is meaningless except when throwing an exception, when it is called with an unexpected combination of properties.

This implementation solves the above problems. If we add a new type property, we must only write a coordinator and give the appropriate precedence order. If we must write a special behavior for combination with other properties, we must only write an appropriate combination coordinator. If we change the order of precedence, we must only modify the `precedence`. Moreover, we do not need to write the same process in many methods.

7. The Verifier as an Aspect

Our framework provides a bytecode verifier using a parser and a type checker. One advantage of aspect-oriented programming is its ability to unify the cross-cutting concern of non-functional features such as verification. We have actually implemented the verifier in this manner.

An overview of our current implementation of the verifier is as follows. **Figure 8** shows a section of the verifier code.

We can divide this into parsing-time verification and type-checking-time verification subsections. The parsing-time verification subsection includes the pointcut `insertLabel` and the `after` advice of `insertLabel`. The

```
public aspect Coordinator {
    declare precedence: BottomCoordinator,
       ...
       ObjectAndPrimitiveCoordinator,
       ObjectTypeCoordinator,
       PrimitiveTypeCoordinator,
       ...
       UntypedCoordinator;
}
public abstract class Property {
    public Property meet(Property p) {
        throw new RuntimeException
          ("unsupported property:"
           + this + ", " + p);
    }
}
public abstract aspect PropertyCoordinator {
    pointcut meet(Property p1, Property p2)
        : execution(
          Property Property+.meet(Property+))
        && target(p1) && args(p2);
}
public class Bottom extends Property {}
public aspect BottomCoordinator
  extends PropertyCoordinator {
    Property around(Property p)
        : meet(Property, p)
        && target(Bottom) {
        return p;
    }
    Property around(Property p)
        : meet(p, Property)
        && args(Bottom) {
        return p;
    }
}
...
public class ObjectType extends Property {}
public aspect ObjectTypeCoordinator
  extends PropertyCoordinator {
    Property around
      (ObjectType p1, ObjectType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        // calculate least upper bounds on types
    }
}
...
public aspect ObjectAndPrimitiveCoordinator
  extends PropertyCoordinator {
    Property around
      (ObjectType p1, PrimitiveType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        return new Untyped();
    }
    Property around
      (PrimitiveType p1, ObjectType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        return new Untyped();
    }
}
```

Fig. 7 Implementation of the type property using AspectJ.

```
public aspect Verifier {
  pointcut insertLabel(Instruction inst, int pc)
    : call(void Instruction
      .insertLabel(Instruction[], int))
    && target(inst) && args(Instruction[], pc);
  after(Instruction inst, int pc)
    throwing (IndexOutOfBoundsException e)
    : insertLabel(inst, pc) {
      throw new VerifyException(
        "The target branch is out of bounds: "
        + pc + ":" + inst);
  }
  pointcut stackUnderFlow()
    : call(Object LinkedList.removeFirst())
    && within(TypeChecker);
  after() throwing (NoSuchElementException e)
    : stackUnderFlow() {
      throw new VerifyException
        ("stack under flow");
  }
  pointcut stackOverFlow(LinkedList stack)
    : call(void LinkedList.addFirst(Object))
    && target(stack)
    && within(TypeChecker);
  before(LinkedList stack)
    : stackOverFlow(stack) {
    int maxStack = ...; // how can we get?
    if(stack.size() >= maxStack) {
       throw new VerifyException
         ("stack over flow:" + maxStack);
    }
  }
}
```

Fig. 8 Implementation of the verifier.

parser calls the method `insertLabel` when it finds a branch instruction. If the branch refers to a location outside the bounds of the code, `insertLabel` throws an `IndexOutOfBoundsException`. Advice of the verifier catches the exception and rethrows a `VerifyException`.

The type-checking-time verification subsection includes the `stackUnderFlow` and `stackOverFlow` parts. The `TypeChecker` class extends our dataflow analyzer framework and uses `LinkedList` in the Java class library as the operand stack.

In this design and implementation, we have found that AspectJ has two problems currently at least.

- **The dependence problem**: AspectJ is not sufficiently expressive to structure aspects in order to avoid to depend on inner structure strongly.
- **The additional information problem**; AspectJ cannot ensure to append additional information safely and generally.

7.1 The Dependence Problem

In our implementation, the verifier depends

strongly on the inner structure of the parser and the type checker. Therefore, not only must we modify the verifier whenever we modify the parser or type checker, but we cannot reuse the verifier with, for example, another instruction set.

This problem is partially solved using aspect structuring, i.e., dividing the verifier into parts that are dependent and independent of instruction sets, and parameterizing the dependent part. However, AspectJ cannot currently separate the verifier in this manner. Abstract pointcuts are useful, but they are insufficient to perform this separation.

Parametric introductions [10], which areinter-type introductions parameterized target classes, may solve this problem partially. This solution is as follows:

(1) Define generic aspects that insert verification code to the abstract parser and type checker.
(2) Parameterize the aspects to the concrete parsers and type checkers using connector aspects.

However, it is successful only in the case that the abstract parser and type checker can be defined sufficiently well, that is, if we cannot extract an abstraction of parsers and type checkers, we also cannot define generic aspects. In this case, we have found that verification code is too heterogeneous to define generic aspects.

7.2 The Additional Information Problem

We designed our framework to separate an analyzer from the target bytecode, i.e., the analyzer should not hold any analyzing state information about the target bytecode; the target bytecode should hold all of this analyzing state information. The operand stack then belongs to the target.

When the list is empty and the method `removeFirst` is called, the `LinkedList` throws a `NoSuchElementException`. Therefore, the pointcut and the advice of `stackUnderFlow` catches the exception and then rethrows a `VerifyException`.

In contrast, implementation of the process of `stackOverFlow` engenders a problem when attempting to retrieve the maximum stack size. The advice of `stackOverFlow` can access the operand stack and this join point. We may extract information about the type checker classes from this join point. However, according to our design policy, the type checker classes hold no code information, such as the maximum stack size.

On the other hand, the operand stack originally does not hold the maximum stack size because it is an instance of `LinkedList` in the Java class library. If we wish to add the maximum stack size to the stack, we must establish the maximum stack size of the list in advance. It is difficult to ensure this setting for general cases.

It may not be possible to provide advice with information only from a pointcut. For example, we cannot provide advice to detect the overflow of the operand stack naturally because its pointcut gives only an operand stack object as a parameter. In addition, the object does not provide a method to retrieve the maximum stack size defined in each method.

These shortcomings may be solved by defining advice and an inter-type field declaration by adding information about the corresponding method to the stack object. However, this is a specific and ad hoc approach. Moreover, this cannot ensure to set the maximum stack size of the operand stack on the all control-flow path.

This problem may be solved in the following way:

- Introduce a rich pointcut that ensure to set additional information on the all control-flow path.
- Compose the rich pointcut from the existing pointcut such as `if` pointcut in AspectJ and/or `pcflow` [13].
- Define the rich pointcut using pointcut description languages such as Refs. 3) and 9)

First, we cannot found such an existing rich pointcut. Second, we cannot compose the rich pointcut from the existing pointcut because there is no pointcut that captures setting code information on the all control-flow path to use it for verification.

Last, we found that Josh [3] is too low expressive to write the rich pointcut. Josh has only basic reflection mechanism. We found defining the rich pointcut need control/data flow analysis mechanism.

8. Assessment

In this section, in order to evaluate the effectiveness of our implementation techniques of AspectJ, we compare two bytecode frameworks and specializers written in Java and AspectJ [22]. Note that both implementation cannot be compared *directly* because we have im-

Table 1 Code size (line number) and file number comparison.

technique name	framework		specializer	
	Java	AspectJ	Java	AspectJ
Direct Extension	676/159	0/0	503/4	1027/9
Abstract Category	–/–	–/–	–/–	–/–
Smart Visitor	2453/162	1571/1	1300/5	143/4
Binary Operation	181/1	506/2	0/0	0/0

plemented the framework and the specializer written in AspectJ not from the ones written in Java but from scratch.

Table 1 shows code size and file number in these specializers and our framework in which we use the proposed techniques. The data of the abstract category is not available because we have not implemented by the abstract category completely, yet.

The direct extension technique in Java is simple visitor applied only to instruction classes. A concern on instruction visitor in framework is spread among all concrete instruction classes, because each instruction class must have a method `accept`. Moreover, because it cannot be applied to other classes, the classes in specializer using it are only 4. On the other hand, the direct extension technique in AspectJ can be applied not only to instruction visitor but also other classes. Plus, it eliminates the code size of both the framework and the specializer.

The smart visitor technique includes the direct extension technique because it is realized by the simple visitor. The visitor concern in the framework written in Java is spread among instructions and its code size tends to be large. On the other hand, in the framework written in AspectJ, the visitor concern is only in one module, and its code size is reduced. The smart visitor technique is more effective in the specializer. The code size of the specializer is eliminated drastically.

The binary operation technique, however, enlarges the code size, though the data flow analyzer in the framework written in AspectJ is more expressive than the one written in Java because of the technique.

9. Related Work

9.1 Bytecode Analyzers

Joeq [21] is an extensible virtual machine and compiler infrastructure. It has many sophisticated features and can be used as a bytecode analyzer framework.

Joeq provides the Visitor framework, enabling a simple analyzer implementation. Joeq can realize an analyzer by overriding the defined methods *in advance* for some situations, such as field accesses in an instance. Therefore, a programmer cannot unify arbitrary methods with the same behavior. On the contrary, our Smart Instruction Visitor can realize an analyzer using pointcuts, which can be defined freely by a programmer without performance penalty.

Joeq also provides a dataflow framework in which the binary operations of properties are defined in the centralized dataflow problem class. Whaley does not show an implementation detail for binary operations. It would be complicated for some analyzers, such as a type analyzer.

Although OVM [18] is specialized for a virtual machine, its design policy can apply a bytecode analyzer. The main advantage of OVM is its memory efficiency; the OVM intermediate representation (OvmIR) uses the Flyweight Pattern [7]. In contrast, our current implementation requires more memory than OVM.

OVM also adopts the Runabout Pattern [8], making it more extensible, but giving it worse execution performance than the Visitor Pattern approach. This tradeoff is unavoidable when using Java and Java-based languages such as AspectJ. OVM focuses on the customization of the intermediate representation, so OVM has opted for extensibility and the Runabout approach. Nevertheless, we choose to optimize performance by using the Visitor approach because the main target of our framework is Java bytecode.

Ideally, our approach should be mixed: in the early stage of development, we should take the Runabout approach. When the specifications of the intermediate representations are almost fixed, we should switch to the Visitor approach. To ease the switch, we will need an automatic code translator to convert from the Runabout to the Visitor.

9.2 Aspect-oriented Design and Implementation

Coady and Kiczales refactor FreeBSD operating system using AspectC [5]. They found and

refactor tangled code of page daemon activation, prefetching, disk quotas, and blocking in device drivers. They also evaluate that the extracted aspect is robust through system evolution. Because their approach is based on existing large system, its evaluation is more scientific than ours. However, because we have built the framework from scratch, we can design the fundamental architecture using the advantages of AspectJ.

Ségura-Devillechaise, et al. have built an aspect-oriented web cache system [19]. They discuss how aspect-orientation refines web caches and prefetching, and propose dynamic native-code-based weaving system. Their aspects are more course-grained than ours, and are attached and detached as modules.

Colyer and Clement have built large-scale middleware for distributed system [6]. They categorize cross-cutting concerns into homogeneous and heterogeneous. They propose several techniques on homogeneous cross-cutting concerns and systematic refactoring process on heterogeneous cross-cutting concerns.

Harbulot and Gurd report separation of concerns in scientific computing [12]. They try to separate computation algorithm and parallelization, especially loop. They also propose aspect-oriented refactoring techniques.

Bodkin and Almaer report many techniques that are found through development of aTrack, which is a web-based bug tracking tool [1]. The techniques is more fine-grained than our techniques. For example, they propose an exception handling technique in AspectJ, which contains only one advice.

9.3 Design Patterns

We can refine our techniques to design patterns for AspectJ if we show their general applicability.

Hannemann [11] discusses how to describe the Gang-of-Four (GoF) design patterns [7] in AspectJ. Although he proposes new implementation of GoF patterns, he does not propose new patterns that cannot be described without aspect orientation.

10. Conclusions and Future Work

We have built a Java bytecode analyzer framework that uses aspects. Thereby, we observed five advantages. Firstly, we realized extensions of elementary objects structurally and maintained type safety and execution efficiency. Secondly, we implemented a bytecode parser that is independent of any single concrete instruction set. Thirdly, we simplified the description of processes for each instruction using the Smart Instruction Visitor based on the stack machine model. Fourthly, we realized binary operations that are simple, extensive, and easy to maintain. Finally, we unified the description of a cross-cutting concern of a wide ranging nonfunctional features such as verification.

Furthermore, we observed that AspectJ currently has two limitations: it is not sufficiently expressive to structuralize aspects deeply on the basis of their inner structure; it does not provide a general approach to write advice that cannot be described with its pointcut only.

In the future, we will build a bytecode translator framework based on aspect-oriented software development. It will allow us to build many applications, including a bytecode-level optimizing compiler.

Acknowledgments This research was supported in part by a grant of Fukuoka project in the Cooperative Link of Unique Science and Technology for Economy Revitalization (CLUSTER) of Ministry of Education, Culture, Sports, Science and Technology (MEXT).

We are grateful to Etsuya Shibayama in Tokyo Institute of Technology, Hidehiko Masuhara in Tokyo University and Naoyasu Ubayashi in Kyushu Institute of Technology for many helpful comments and stimulating discussions.

References

1) Bodkin, R. and Almaer, D.: aTrack: An Enterprise Bug Tracking System using AOP, `https://atrack.dev.java.net/` (2004).
2) Chiba, S.: Load-Time Structural Reflection in Java, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2000*), Sophia Antipolis and Cannes, France, pp.313–336, Springer-Verlag (2000).
3) Chiba, S. and Nakagawa, K.: Josh: an open AspectJ-like language, *Proc.International Conference on Aspect-Oriented Software Develop-Ment* (*AOSD 2004*), Lancaster, UK, pp.102–111 (2004).
4) Choi, J.-D., Grove, D., Hind, M. and Sarkar, V.: Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs, *Workshop on Program Analysis For Software Tools and Engineering*, pp.21–31 (1999).
5) Coady, Y. and Kiczales, G.: Back to the Future: A Retroactive Study of Aspect Evolu-

tion in Operating System Code, *Proc. International Conference on Aspect-Oriented Software DevelopMent* (*AOSD 2003*), Boston, Massachusetts., pp.50–59, ACM Press (2003).
6) Colyer, A. and Clement, A.: Large-scale AOSD for Middleware, *Proc. International Conference on Aspect-Oriented Software DevelopMent* (*AOSD 2004*), Lancaster, UK, pp.56–65, ACM Press (2004).
7) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
8) Grothoff, C.: Walkabout revisited: the Runabout, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2003*), Darmstadt, Germany, pp.103–124, Springer-Verlag (2003).
9) Gybels, K. and Brichau, J.: Arranging Language Features for More Robust Pattern-based Crosscuts, *Proc. International Conference on Aspect-Oriented Software DevelopMent* (*AOSD 2003*), Boston, Massachusetts, pp.60–69, ACM Press (2003).
10) Hanenberg, S. and Unland, R.: Parametric introductions, *Proc. International Conference on Aspect-Oriented Software DevelopMent* (*AOSD 2003*), Boston, Massachusetts, pp.80–89, ACM Press (2003).
11) Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, *Proc.ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA 2002*), Seattle, Washington, USA, pp.161–173, ACM Press (2002).
12) Harbulot, B. and Gurd, J.R.: Using AspectJ to Separate Concerns in Parallel Scientific Java Code, *Proc. International Conference on Aspect-Oriented Software DevelopMent* (*AOSD 2004*), Lancaster, UK, pp.122–131, ACM Press (2004).
13) Kiczales, G.: The Fun Has Just Begun, *Keynote talk at Proceedings of the International Conference on Aspect-Oriented Software DevelopMent* (*AOSD 2003*), Boston, Massachusetts. (2003).
14) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2001*), Budapest, Hungary, pp.327–353, Springer-Verlag (2001).
15) Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Iriwin, J.: Aspect-Oriented Programming, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 1997*), Jyväskylä, Finland, Akşit, M. and Matsuoka, S.(eds.), Vol.1241, pp.220–242 (1997).
16) Leroy, X.: Java Bytecode Verification: Algorithms and Formalizations, *Journal of Automated Reasoning*, Vol.30, No.3–4, pp.235–269 (2003).
17) Nielson, F., Nielson, H. R. and Hankin, C.: *Principles of Program Analysis*, Springer, Berlin (1999).
18) Palacz, K., Baker, J., Flack, C., Grothoff, C., Yamauchi, H. and Vitek, J.: Engineering a Customizable Intermediate Representation, *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators* (*IVME'03*), pp.67–76, ACM Press (2003).
19) Ségura-Devillechaise, M., Menaud, J.-M., Muller, G. and Lawall, J. L.: Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution, *Proc. International Conference on Aspect-Oriented Software DevelopMent* (*AOSD 2003*), Boston, Massachusetts, pp.110–119, ACM Press (2003).
20) Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E. and Co, P.: Soot: a Java Optimization Framework, available at `http://www.sable.mcgill.ca/soot/` (1999).
21) Whaley, J.: Joeq: A Virtual Machine and Compiler Infrastructure, *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators* (*IVME'03*), pp.58–66, ACM Press (2003).
22) Yamazaki, S., Kando, T., Matsumoto, M., Nakanishi, T., Kitasuka, T. and Fukuda, A.: Logic-based Binding Time Analysis for Java using Reaching Definitions, *IPSJ Transaction on Programming*, Vol.46, No.SIG1(PRO24), pp.121–133 (2005).
23) Yamazaki, S., Matsumoto, M., Nakanishi, T., Kitasuka, T. and Fukuda, A.: Aspect-Oriented Design and Implementation in Java Bytecode Analyzer Framework, *Proc. 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (*ACP4IS*), Lancaster, UK (2004).

(Received July 2, 2004)
(Accepted October 11, 2004)

Susumu Yamazaki was born in 1972. He received his B.E. degree in metallurgy from Tokyo Institute of Technology, Japan, in 1995 and his M.S. degree in information science from Tokyo Institute of Technology, Japan, in 1997. He took a doctor's course of Tokyo Institute of Technology from 1997 to 2003. He was a researcher of Graduate School of information science and engineering, Tokyo Institute of Technology, Japan in 2003. Since 2003, he has worked for Fukuoka Laboratory for Emerging and Enabling Technology of SoC, Fukuoka Industry, Science and Technology Foundation. His research interests include compilers, systems and development methodologies. He is a member of ACM, IPSJ and JSSST.

Michihiro Matsumoto received the B.S. degree in mathematics from Kyoto University, Japan, in 1990; and the Master's degree and Ph.D. degree in information science from Japan Advanced Institute of Science and Technology, Japan, in 1998 and 2002, respectively. From 1990 to 2003, he was a software engineer, a researcher or a system engineer of PFU Limited. Since 2003, he has been a researcher of Fukuoka Laboratory for Emerging and Enabling Technology of SoC, Fukuoka Industry, Science and Technology Foundation. His research interests include development methodologies for embedded systems.

Tsuneo Nakanishi received the B.E. degree in communication engineering from Osaka University, Japan, in 1993; and the M.E. and D.E. degrees in information science from Nara Institute of Science and Technology, Japan, in 1995 and 1998, respectively. From 1998 to 2002, he was an assistant professor of Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2002, he has been an associate professor of Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan. His research interests include development enviroment for embedded systems.

Teruaki Kitasuka received the B.E. degree in information science from Kyoto University, Japan, in 1993 and M.E. degree in information science from Nara Institute of Science and Technology, Japan, in 1995. From 1995 to 2001, he worked for the Sharp Corporation, where he developed the personal computer. Since 2001, he has been a research associate of Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan. His research interests include mobile computing, embedded systems, parallel and distributed systems, compiler, and computer architecture.

Akira Fukuda received the B.E., M.E., and Ph.D. degrees in computer science and communication engineering from Kyushu University, Japan, in 1977, 1979, and 1985, respectively. From 1977 to 1981, he worked for the Nippon Telegraph and Telephone Corporation, where he engaged in research on performance evaluation of computer systems and the queueing theory. From 1981 to 1991 and from 1991 to 1993, he worked for the Department of Information Systems and the Department of Computer Science and Communication Engineering, Kyushu University, Japan, respectively. In 1994, he joined Nara Institute of Science and Technology, Japan, as a professor. Since 2001, he has been a professor of Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan. His research interests include embedded systems, system software (operating systems, compiler, and runtime systems), mobile computing, parallel and distributed systems, and performance evaluation. He received 1990 IPSJ (Information Society of Japan) Research Award and 1993 IPSJ Best Author Award. He is currently the chair of SIG System Evaluation in IPSJ. He is a member of ACM, IEEE Computer Society, IEICE, IPSJ, and the Operations Research Society of Japan.