Bytecode rewriting in Tom

Emilie Balland, Pierre-Etienne Moreau, Antoine Reilles

Bytecode rewriting in Tom

Emilie Balland, Pierre-Etienne Moreau and Antoine Reilles

UHP & LORIA, INRIA & LORIA and INPL & LORIA
{Emilie.Balland,Pierre-Etienne.Moreau,Antoine.Reilles}@loria.fr

LORIA
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex France

Abstract

In this paper, we present a term rewriting based library for manipulating Java bytecode. We define a mapping from bytecode programs to algebraic terms, and we use Tom, an extension of Java that adds pattern-matching facilities, to describe transformations. An originality of Tom is that it provides a powerful strategy language to express traversals over trees and to control how transformation rules are applied. To be even more expressive, we use CTL formulae as conditions and we show how their satisfiability can be ensured using the strategy formalism. Through small examples, we show how bytecode analysis and transformations can be defined in an elegant way. In particular, we outline the implementation of a ClassLoader parameterized by a security policy that restricts file access.

1 Motivations

Bytecode transformation is an example of a late code modification technique, targeting an executable program after compilation. Traditional late code modification systems suffer from lacking certain useful information available only to source code. However, Java classfiles keep symbolic information that makes this approach viable, and thus enables a wide range of transformations [6]. Bytecode transformations consist in adding or removing fields, methods, or constructors; changing the superclass, changing the types or signatures of fields or methods, redirecting method invocations to new targets (for example to call secured methods), and inserting new code into methods. This approach can be used to specify security properties or to adapt code to an embedded JVM.

There exist several libraries for manipulating Java bytecode, among them SERP [12], BCEL [1] and ASM [2] are the most well-known. Although they are powerful, a deep knowledge of the API may be needed to use them effectively. In this paper we propose to introduce an abstraction level, based on terms and rewriting, to make the definition of high-level transformations and analysis easier. Using the notion of algebraic view, we have extended the ASM library such that a bytecode program can be seen as a term. This gives us the possibility to directly express transformation rules without knowing the API, and thus to reduce the gap between

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the user's wishes and the language expressiveness. This approach can be considered similar to a domain specific language (DSL) for bytecode transformations.

In this article, we consider the language Tom, an extension which adds pattern matching and strategic programming primitives to existing languages such as Java, and we show how it can be extended to easily express bytecode transformations. Indeed, pattern-matching is directly related to the structure of objects and therefore is a very natural construct, commonly found in functional languages, to recognize a pattern and retrieve information. In addition, Tom provides a powerful strategy language that can be used to control rule applications. The separation of concerns between rules and strategies introduces well founded constructs and allows us to reason about transformations. This improves reusability, expressiveness, and thus helps to avoid introducing programming errors when implementing a complex transformation. These two features are particularly well-suited to describe transformations of structured entities like, for example trees, terms, or programs.

After presenting the Tom language and its connexion to ASM in Section 2, we show in Section 3 why it is apt to express bytecode analysis. In this section we consider a simple optimization example to outline our ideas. In particular, we introduce Tom strategies as a very elegant way to express temporal properties on the code of a method. In Section 4 we give an example of bytecode transformation that ensures a given security policy.

2 Java classes as terms of Tom language

2.1 Representing a Java bytecode program

The BCEL library has been introduced to modify Java bytecode programs. Given a class, it provides an object representation in memory. The ASM framework [2] is a similar tool, which uses a different model to avoid constructing the whole representation of the bytecode class in memory at once. ASM's design is based on an event-driven model and uses a visitor design pattern to avoid representing visited structures with objects. Visitors receive events for particular pieces of the structure from the event generator corresponding to the ClassReader class, which knows how to parse Java bytecode from existing classes and how to fire appropriate events to the underlying visitors.

In our setting, the notion of *term* is essential because this is the only data-structure that can be handled by a rewriting rule. For this reason, we developed Gom [8], a generator of typed tree structures. Given an algebraic signature, it generates a Java implementation that is efficient in time and space.

In order to represent bytecode programs, we have defined an algebraic signature that allows us to represent any bytecode program by a typed term. Given a Java class, we use ASM to read the content and build an algebraic representation of the complete Java class. This approach is similar to BCEL. This permits multi-pass or global analysis. However, by using hash-consing techniques, we obtain a data structure with maximal sharing, ensuring a minimal memory footprint.

For example, to represent a Java class as an algebraic term we have defined the following signature:

```
module Bytecode
imports int long float double String
abstract syntax
TClass = Class(info:TClassInfo, fields:TFieldList,
               methods:TMethodList)
...
TMethodList = MethodList(TMethod*)
TMethod = Method(info:TMethodInfo, code:TMethodCode)
TMethodCode = MethodCode(instructions:TInstructionList,
                         localVariables:TLocalVariableList,
                         tryCatchBlocks:TTryCatchBlockList)
...
TInstructionList = InstructionList(TInstruction*)
TInstruction = Nop()
             | Iload(var:int)
             | Ifeq(label:TLabel)
             | Invokevirtual(owner:String, name:String,
                             methodDesc:TMethodDescriptor)
 ...
```

The real signature contains more than 250 different constructors. Due to lack of space we cannot list them in detail. The given signature shows that a class is represented by a constructor `Class`, which contains information such as name, packages, and imports. It also contains a list of fields and a list of methods. The latter is encoded using an associative operator `MethodList` whose arity is not fixed. As illustrated below, it is used to model lists and will be useful to describe the search of one or several elements in the list. Similarly, a list of instructions is represented by the associative operator `InstructionList`. A method contains an `info` part and a `code` part. The `code` part is mainly composed by local variables and a list of instructions. Each bytecode instruction is represented by an algebraic constructor: `Nop`, `Iload`, *etc.*

Given this signature, Gom generates an efficient Java implementation (one class for each constructor) that allows us to represent and manipulate the representation of a Java class. As illustrated in Figure 1, given a Java class `C`, ASM is used to build an algebraic representation `TC`. Using the expressive rule based framework provided by Tom, this term can be rewritten into another one (`TC'`), which is later being dumped back to a new Java class `C'`.

2.2 Retrieving information by pattern-matching

Tom is a language extension that adds pattern matching primitives to existing imperative languages such as C and Java. Pattern matching is usually found in functional and logic languages, where it is used to analyse the composition of aggregated data, and express transformations of those structured data. The main originality of the Tom system is its language and data-structure independence [7]. From an implementation point of view, it is a compiler which accepts different *host languages* and whose compilation process consists in translating the matching constructs into

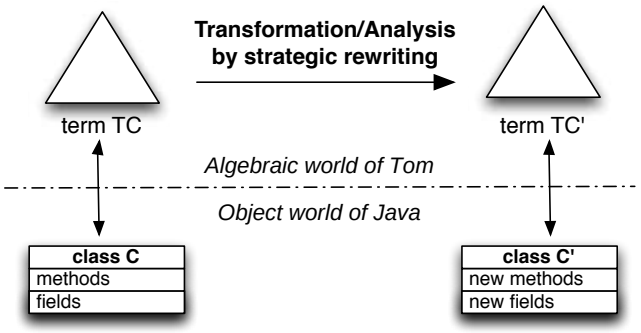

Fig. 1. Bytecode transformations in Tom

the underlying native language. Since Gom generates Java implementations of terms, in the following, we consider Java as the host language.

On the practical side, Tom provides two interesting features: a "`%match`" construct to match objects and a "`‘`" construct to build objects. The `%match` construct adds pattern matching facilities similar to functional languages. The action part is written in Java, providing flexibility. For example, considering the algebraic signature given previously, we can express the function that modifies the indexes of an `Istore` or an `Iload` in the following way:

```
Instruction updateIndex(Instruction i, int oldI, int newI) {
  %match(i) {
    Istore(x) -> { if(‘x==oldI) return ‘Istore(newI); }
    Iload(x) -> { if(‘x==oldI) return ‘Iload(newI); }
  }
}
```

Note the use of "`‘`" to build a new term such as `Istore(newI)` or to access a variable instantiated by pattern matching such as `x`. The "`‘`" construct can be used anywhere a Java expression is allowed.

In addition to syntactic matching (*a la* ML), the language provides associative matching with neutral element (also known as list-matching). This is particularly useful to model the exploration of a search space and to perform list based transformations. For example, let us consider the statement `t=‘InstructionList(Istore(3),Istore(2),Iload(3));` which builds the term representing a sequence of three bytecode instructions.

To illustrate the expressiveness of list-matching we will first define a function that checks whether an `Istore` is performed by a list of instructions. This can be expressed as follows:

```
boolean hasIstore(InstructionList l) {
  %match(l) {
    InstructionList(a*,Istore(index),b*)-> { return true; }
  }
  return false;
}
```

In this example, *list variables* annotated by a `*` are instantiated by a (possibly empty) list. Therefore, for the sequence `t`, `Istore(index)` will be found in first position, and the Java variable `index` will be instantiated by the integer `3`. The variables `a` and `b` are respectively instantiated by the empty list `InstructionList()` and `InstructionList(Istore(2),Iload(3))`.

Non linearity can be used to search for a variable that is both stored and loaded:

```
boolean hasStoreAndLoad(InstructionList l) {
  %match(l) {
    InstructionList(_*,Istore(index),_*,Iload(index),_*)-> {
      return true;
    }
  }
  return false;
}
```

Note the multiple use of `index` to enforce that a found `Iload` corresponds to a previous `Istore`. Note also the use of `_*` to denote anonymous variables. When applied to `t`, the answer is

3 Bytecode Analysis and Transformation

As an example of transformation, we want to eliminate store instructions which are not necessary in a program. Considering the following bytecode program:

```
Istore(3);Istore(2);Iload(3);Istore(2);Iload(2)
```

We want to detect every store in the control flow graph such that there is no load instruction with the same index before another store. In the previous example, the first instruction `Istore(2)` can be removed.

In [5], D. Lacey has introduced an original method for describing transformations of programs based on rewriting and computational tree logic [3](CTL). This formalism inspired us to offer in Tom facilities to express CTL formulae on terms. In this section we will present the strategy language provided by Tom and show how any CTL formula can be translated into a strategy expression. The evaluation mechanism of the strategy language gives us a procedure to check that a CTL formula is true: an unfailing terminating strategy ensures that the CTL formula is true.

3.1 CTL logic for bytecode analysis

As well as usual logical operators like $\neg$ or $\wedge$, CTL formulae can be composed of temporal operators. These operators are composed by one path operator followed by a state operator [4].

- *Path operators* state whether the formula ϕ should hold for all possible execution paths of the system ($A\phi$) or only for at least one execution path ($E\phi$). Those operators are specific to CTL.

- *State operators* come from LTL (linear temporal logic), and state facts about the future of paths:
  - $X\phi$ – Next: ϕ has to hold at the next state,
  - $G\phi$ – Globally: ϕ has to hold on the entire subsequent path,
  - $F\phi$ – Finally: ϕ eventually has to hold (somewhere on the subsequent path),
  - $\phi U\phi'$ – Until: ϕ has to hold until at some position ϕ' holds. This implies that ϕ' will be verified in the future.

For example, invariants in the code are represented by the CTL formula $AG(p)$ where p is the predicate we want to be always verified. Fatality properties can be defined using the combination of A and F. For a method, we can for example verify that it always returns a result if it is not a void method. That means that if the predicate *IsReturn* verifies that an instruction is of type `Return`, the corresponding CTL formula is $AF(IsReturn)$.

Considering again our initial example, the search of unused stores can be translated into the formula $IsStore(i) \wedge AX(AU(\neg IsLoad(i), IsStore(i)))$ where $IsLoad(i)$ is the predicate that is true if the current node of the control flow graph corresponds to a `Load(i)` (similar for $IsStore(i)$). This example shows that CTL formulae are an easy way to define bytecode analysis. We will explain how the Tom language is well-suited for defining such specifications. Indeed, strategies can be clearly interpreted in term of CTL formulae.

3.2 From CTL formulae to Strategy expressions

In the rewriting community, the notion of strategy has been introduced to describe, in an elegant way, how to apply a set of rules. They add higher-order features to first-order term rewriting rules.

An elementary strategy can be either a *transformation rule*, the `Identity`(does nothing), or a `Fail`(always fails): they are the basic ingredients. In our system, an elementary strategy is type-preserving and is defined by extending a default strategy:

```
%strategy IntToDouble() extends Fail() {
  visit Instruction {
    Istore(i) -> { return `Dstore(i); }
    Iload(i) -> { return `Dload(i); }
  }
}
```

An elementary strategy contains an implicit `%match`: when applied to a node of sort `Instruction`, a transformation is performed if a pattern matches the node. Otherwise, the default strategy is applied (`Fail` in the example).

On top of elementary strategies, more complex strategies can be built, involving basic combinators such as `Sequence(s1,s2)`, `Choice(s1,s2)`, `All(s)`, `One(s)`, *etc.* as presented in [10,11].

The basic combinators can be separated in two categories: the traversal combinators (`All`, `One`) and the control combinators (`Sequence`, `Choice`, `Not`,*etc.*).

- *Traversal combinators* describe how to rewrite in a direct subterm:

  - When applied to a term $f(t_1, \ldots, t_n)$, `All(s)` returns $f(t'_1, \ldots, t'_n)$, where t'_i corresponds to the application of `s` to t_i. `All(s)` fails if it fails on at least one t_i.
  - `One(s)` is similar to `All(s)` but `s` is applied to only one subterm t_i. It fails if `s` cannot be applied on at least one subterm (i.e. it fails when applied to a constant).
- *Control combinators* describe how to compose different strategies, depending on their results:
  - `Sequence(s1,s2)` denotes a sequential composition of `s1` and `s2`: it fails if `s1` or `s2` fails, otherwise it results in the application of `s2` on the result of `s1`.
  - `Choice(s1,s2)` denotes a left-to-right choice: if `s1` succeeds, the result is returned, otherwise `s2` is applied.
  - `Not(s)` performs `Identity` if `s` fails and fails otherwise.
  - `If(s1,s2,s3)` starts by applying `s1`. If it succeeds, it returns `s2`, otherwise `s3`. The result obtained with `s1` is just used for the test as a conditional expression.

By combining elementary strategies and basic combinators, it becomes possible to define higher-level constructs. For example, the fix-point operator can be expressed by `Repeat(s)` $\triangleq \mu$`x.Choice(Sequence(s,x),Identity())`, where μ denotes a recursion operator (similar to `rec` in ML), `x` a variable, and `s` a parameter of the strategy. This strategy will apply the strategy `s` repeatedly until it fails, and then return the last result obtained before failure, thus `Repeat` will never fail.

3.3 Verification of CTL formulae by Strategies

In CTL there is a minimal set of operators. All CTL formulae can be transformed to use only those operators. One minimal set of operators is $\{false, \vee, \neg, EG, EU, EX\}$. Other operators can be defined with this set:

$$\begin{aligned} EF\phi &= \neg EU(true, \neg\phi) \\ AX\phi &= \neg EX(\neg\phi) \\ AG\phi &= \neg EF(\neg\phi) \\ AF\phi &= \neg EG(\neg\phi) \\ AU(\phi_1, \phi_2) &= \neg EU(\neg\phi_1, \neg(\phi_1 \vee \phi_2)) \vee EG(\neg\phi) \end{aligned}$$

The classical strategy operators can be combined in an interesting way to obtain temporal strategy aliases. The main idea is to represent a predicate p as a basic strategy s (defined by a `%strategy`) and the CTL operators by strategy aliases. Instead of using a model-checker like SPIN to verify a temporal formula on the method code, a specific strategy is applied to the list of instructions. If the visit fails, the formula is false. Otherwise, it is considered as true.

The boolean values *false* and *true* are represented by the strategies `Fail` and `Identity`. If we only consider the minimal set of operators, they can be defined using Tom strategy combinators. We consider the interpretation function $[\![\ ]\!]$ from formulae to strategies defined as follows:

$[\![false]\!]$ ≡ `Fail`

$[\![p_1 \vee p_2]\!]$ ≡ `Choice(`$[\![p_1]\!]$`,`$[\![p_2]\!]$`)`

$[\![\neg p]\!]$ ≡ `Not(`$[\![p]\!]$`)`

$[\![EG(p)]\!]$ ≡ μ`x.Sequence(`$[\![p]\!]$`,If(One(Identity),One(x),Identity))`

$[\![EU(p_1,p_2)]\!]$ ≡ μ`x.Choice(`$[\![p_2]\!]$`,Sequence(`$[\![p_1]\!]$`,One(x)))`

$[\![EX(p)]\!]$ ≡ `One(`$[\![p]\!]$`)`

In the example of unused store search in Section 3.1, three predicates were used. For each, a basic strategy can be defined using the construction `%strategy` of Tom. To represent indexes, strategies cannot directly be parameterized by an Integer because integers are immutable and we need to modify its value during the traversal. A solution consists in using an Integer wrapper.

`IsLoad(w)` succeeds only if the term visited is of type Load and when its index is equal to the index of wrapper `w`.

```
%strategy IsLoad(w:IntWrapper) extends Fail() {
  visit Instruction {
    c@(Iload|Lload|Fload|Dload|Aload)(i) -> {
      if(w.intValue().equals(‘i)) return ‘c;
    }
  }
}
```

In this example, `@` corresponds to an annotation mechanism for keeping a reference to a matched subterm. Here, the variable `c` is instantiated with the redex and used in the right hand side part.

The code of `IsStore(w)` is similar to `IsLoad(w)`. `IsStore(w)` succeeds only if the term visited is of type Store and if its index is equal to the index of the wrapper `w`. `IndexStore(w)` succeeds only if the term visited is of type Store and set the corresponding index in the wrapper `w`.

```
%strategy IndexStore(w:IntWrapper) extends Fail() {
  visit Instruction {
    c@(Istore|Lstore|Fstore|Dstore|Astore)(i) -> {
      w.setValue(‘i); return ‘c;
    }
  }
}
```

We can now define our analysis with the following composed strategy:

```
IntWrapper w = new IntWrapper(0);
Strategy storeNotUsed =
  ‘And(IndexStore(w),AX(AU(Not(IsLoad(w)),IsStore(w))));
```

```
/* Use storeNotUsed to remove unused stores in ins */
`BottomUp(If(storeNotUsed,Remove(),Identity())).visit(ins);
```

where `ins` is a variable of type `InstructionList` and `BottomUp` is a traversal from leaves to root. This traversal can be defined as `BottomUp(s)` $\triangleq$ μ`x.Sequence(All(x),s)`. The execution of this code will remove all unused store instructions due to the elementary strategy `Remove()` that removes the first instruction of the list. You can notice that `storeNotUsed` is not defined as an elementary strategy (construction `%strategy`) but by composition.

Simulation of control flow by Strategies

Up to now, we have just defined traversals on the list of instructions without considering the control flow. Obviously, the use of strategies as temporal formulae is only interesting in a control flow context. A first idea is to construct explicitly the control flow graph from the list of instructions but the memory cost in case of complex methods cannot be disregarded. Our suggestion is to use strategies in order to simulate the control flow during the traversal of the list of instructions. In this way, a memory representation of the control flow graph is not necessary. In a list of instructions, the control transits from an instruction to the following in the list and we do not care of `Goto` instructions or possible exceptions that can disturb the control flow.

In the Tom language, the rules and the control are completely separated so an alternative for representing control flow graphs (CFG) is to use the control to indicate what is the possible following instruction. We have seen in the previous section that to apply a strategy to children, there exist two generic congruence operators `All` and `One`. We can redefine these two combinators such that they behave as if the children were the following instructions in the CFG.

For example, suppose we have the following Java code, and the corresponding bytecode:

```
int i = 0;
while (i < 100) {
  i++;
}
```

```
0  Iconst_0()
1  Istore_1()
2  Goto(8)
5  Iinc(1,1)
8  Iload_1()
9  Bipush(100)
11 If_icmplt(5)
```

The `All` strategy is adapted for a CFG run, the full definition is given in Figure 3. For example, the `Goto` instruction has one child with respect to the control flow graph (the instruction corresponding to the label). An `If_XX` instruction has two children: the one which satisfies the expression, and the one that does not. Figure 2 corresponds to the traversal of the bytecode program above.

This strategy is parameterized by a map that associates to every label the sublist of instructions that starts with the labelled instruction. To simplify, we do not

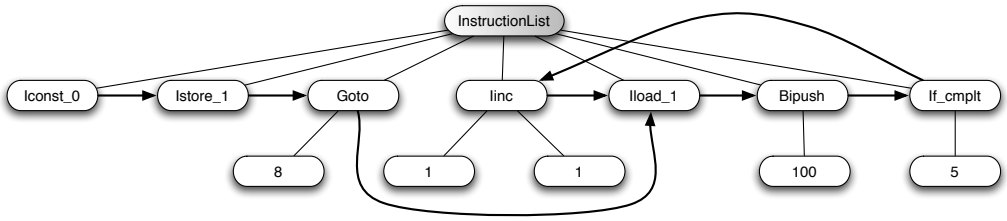

Fig. 2. Example of `All` behaviour

```
%strategy All(s:Strategy, m:Map) extends Identity() {
  visit TInstructionList {
    c@InstructionList((Goto|Jsr)(label), tail*) -> {
      s.visit(m.get(‘label));
      return ‘c;
    }

    c@InstructionList((Ifeq|Ifne|...)(label), tail*) -> {
      s.visit(m.get(‘label));
      s.visit(‘tail*);
      return ‘c;
    }

    c@InstructionList((TableSw|LookupSw)
        [labels=LabelList(_*, label, _*)], tail*) -> {
      s.visit(m.get(‘label));
    }

    c@InstructionList((TableSw|LookupSw)[dft=default], tail*) -> {
      s.visit(m.get(‘default));
      s.visit(‘tail*);
      return ‘c;
    }

    c@InstructionList((Ireturn|...|Return())(label), tail*) -> {
      return ‘c;
    }

    // Default case: Visit the next instruction.
    InstructionList(_, tail*) -> {
      s.visit(‘tail*);
    }
  }
}
```

Fig. 3. `All` strategy specific to bytecode CFG

consider in the definition of the `All` the case for exception handling, but it is resolved in a similar way. It can be interesting in the future to use this formalism to associate to every bytecode instruction the corresponding frame with the meaning of the current state.

4 Secure class loading by bytecode rewriting

In this section, we present an application of bytecode rewriting for secure class loading by redirecting method invocations to new targets. In particular, we will enforce the use of a safe API for file accesses. The use of defensive class loading instead of defensive virtual machine is argued in [9].

This is done by defining a ClassLoader, which will automatically verify whether the code of the class to load contains unsafe pieces of code (in our example we are looking for statements like `new FileReader(nameFile).read()` and replace them by a call to a secure method).

The core of our program is the class `SecureClassLoader` which is a subclass of the standard Java ClassLoader. This class contains Tom code. Like in previous examples, we define a strategy containing transformation rules:

```
%strategy FindFileAccess() extends Identity() {
  visit TInstructionList {
    /* match new FileReader(nameFile).read() */
    InstructionList(before*,
      New("java/io/FileReader"),
      Dup(),
      Aload(number),
      Invokespecial[owner="java/io/FileReader",name="<init>"],
      Invokevirtual[owner="java/io/FileReader",name="read"],
      Pop(),
      after*) -> {
      /* replace it with new SecureAccess(nameFile).sread() */
      return
        `InstructionList(before*,
            New("SecureAccess"),
            Dup(),
            Aload(number),
            Invokespecial("SecureAccess","<init>"),
            Invokevirtual("SecureAccess","sread"),
            Pop(),
            after*);
    }
  }
}
```

The variables `before*` and `after*` are instantiated by a list of instructions. The terms between the variable-stars (`New(...)`, `Dup()`,...) are the Gom terms corresponding to bytecode representation of the `new FileReader(nameFile).read()`

statement. We replace them by the instructions corresponding to the method `sread` of the class `SecureAccess`.

This rule obviously does not capture all semantically equivalent sequences, as for example when the instantiation of the `FileReader` and the call to `read` are separated by some code. Such a situation could be handled by the search of a `read` call following the control flow using the *EF* strategy.

Using Tom list-matching, we apply this strategy for all the methods of the given class:

```
public TClass fileAccessVerify(TClass aclass) {
  TMethodList methods = aclass.getmethods();
  TMethodList secureMethods = `MethodList();
    %match(methods) {
      MethodList(_*, x, _*) -> {
        TInstructionList ins = `x.getcode().getinstructions();
        TInstructionList secureInstList =
          `TopDown(FindFileAccess()).apply(ins);
        TMethodCode secureCode =
          `x.getcode().setinstructions(secureInstList);
        TMethod secureMethod = `x.setcode(secureCode);
        secureMethods = `MethodList(secureMethods*,secureMethod);
      }
   }
   return aclass.setmethods(secureMethods);
}
```

The `getmethods` and `setmethods` methods are defined in the Tom library and are used to get and set methods for a given TClass. Similarly `getcode` and `setcode`, applied to a term of sort TMethod, get and set code, while `getinstruction` and `setinstruction`, applied to a term of sort TMethodCode, get and set code instructions. TopDown is a high-level strategy which traverses a term starting from the root, and is defined using a combination of the elementary strategies described in Section 3 as `TopDown(s)` $\triangleq \mu$`x.Sequence(s,All(x))`.

The method `fileAccessVerify` will be called by the transform method:

```
public byte[] transform(String file) {
    BytecodeReader br = new BytecodeReader(file);
    TClass c = br.getTClass();
    c = fileAccessVerify(c);
    BytecodeGenerator bg = new BytecodeGenerator();
    return bg.toBytecode(c);
}
```

This method illustrates the transformation process described in Figure 1. After getting the Gom term of type TClass from the class file given as parameter, we apply the transformation using `BytecodeReader` and return the array of bytes corresponding to the transformed class, generated by `BytecodeGenerator`.

In our custom class loader `SecureClassLoader`, the only method we redefine

is `loadclass`. Our transformation is applied to the bytecode before the standard `ClassLoader defineClass` method is called, which converts an array of bytes into an instance of a class:

```
public synchronized Class loadClass(String name)
 throws ClassNotFoundException {
    byte[] scode = transform(name);
    Class sClass = defineClass(name,scode, 0, scode.length);
    return loadClass(name,true);
}
```

Using this class loader, we can make sure no class using the unchecked file access method can be loaded, but instead will be transformed at load time to use the safe wrapper class. For simplicity sake, we have only presented the search of the `new FileReader(nameFile).read()` pattern. Analysing the control flow graph is necessary to find the creation of a `FileReader` followed by a call to the method `read()`.

5 Conclusion

We have presented a library that enables bytecode transformations by strategic rewriting. This library allows us to view a Java class as an algebraic term, formed by the tree representation of the bytecode. Transformations can then be described as rewriting rules, using strategies to control their application and explore bytecode expressions.

Moreover, we have demonstrated how those strategies can be used to examine the control flow graph of a method. A particularity of this technique is that it is not necessary to build the control flow graph in memory in order to traverse it. We then show that it is possible to encode the verification of temporal logic conditions over this control flow graph using strategy operators.

Finally, we illustrate the use of this library in the context of defensive execution of classes. We define a specific class loader that redirects classic I/O functions to a safe API to access files. The class loader uses bytecode rewriting to ensure only secured classes are loaded in the JVM. This transformation is defined in an abstract and concise way, thus leading to improve the confidence we can place in the code, by reducing the gap between the transformation to describe and the actual code.

This work is a first attempt to express bytecode transformations with a more abstract approach. A next step will be to better integrate the definition of rewrite rules conditionned by temporal properties [5] in the language, and apply these techniques to program analysis.

Acknowledgments: We sincerely thank Jeremie Delaitre for the initial development of bytecode rewriting in Tom and Pauline Kouzmenko for her work on the secure loading. We also would like to thank the anonymous referees for their valuable remarks and suggestions.

References

[1] *BCEL, Byte Code Engineering Library*, http://jakarta.apache.org/bcel/.

[2] Bruneton, É., R. Lenglet and T. Coupaye, *ASM: a code manipulation tool to implement adaptable systems*, in: *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, 2002.

[3] Clarke, E. M., E. A. Emerson and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. Program. Lang. Syst. **8** (1986), pp. 244–263.

[4] Huth, M. and M. Ryan, "Logic in Computer Science: modelling and reasoning about systems (second edition)," Cambridge University Press, 2004.

[5] Lacey, D., "Program Transformation using Temporal Logic Specifications," Ph.D. thesis, Oxford University Computing Laboratory (2003).

[6] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1999, second edition edition.

[7] Moreau, P.-E., C. Ringeissen and M. Vittek, *A Pattern Matching Compiler for Multiple Target Languages*, in: G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, LNCS **2622** (2003), pp. 61–76.

[8] Reilles, A., *Canonical abstract syntax trees*, in: *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*, 2006, to appear.

[9] Rudys, A. and D. Wallach, *Enforcing Java Run-Time Properties Using Bytecode Rewriting*, in: *Proceedings of the International Symposium on Software Security*, 2002.

[10] Visser, E., Z.-e.-A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, in: *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming* (1998), pp. 13–26.

[11] Visser, J., *Visitor combination and traversal control*, in: *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2001), pp. 270–282.

[12] White, A., *Serp*, http://serp.sourceforge.net/.