Rewriting Approximations for Fast Prototyping of Static Analyzers

Yohan Boichut, Thomas Genet, Thomas Jensen, Luka Le Roux
Université de Rennes 1 and INRIA and CNRS

IRISA, Campus de Beaulieu, F-35042 Rennes, France
{boichut,genet,jensen,leroux}@irisa.fr

Abstract. This paper shows how to construct static analyzers using tree automata and rewriting techniques. Starting from a term rewriting system representing the operational semantics of the target programming language and given a program to analyze, we automatically construct an over-approximation of the set of reachable terms, i.e. of the program states that can be reached. The approach enables fast prototyping of static analyzers because modifying the analysis simply amounts to changing the set of rewrite rules defining the approximation. A salient feature of this approach is that the approximation is correct by construction and hence does not require an explicit correctness proof. To illustrate the framework proposed here on a realistic programming language we instantiate it with the Java Virtual Machine semantics and perform class analysis on Java bytecode programs.

1 Introduction

The aim of this paper is to show how to combine rewriting theory with principles from abstract interpretation in order to obtain a fast and reliable methodology for implementing static analyzers for programs. Rewriting theory and in particular reachability analysis based on tree automata has proved to be a powerful technique for analyzing particular classes of software such as cryptographic protocols [11, 8, 12]. In this paper we set up a framework that allows to apply those techniques to a general programming language. Our framework consists of three parts. First, we define an encoding of the operational semantics of the language as a term rewriting system (TRS for short). Second, we give a translation scheme for transforming programs into rewrite rules. Finally, an over-approximation of the set of reachable program states represented by a tree automaton is computed using the tree automata completion algorithm [8]. In this paper, we instantiate this framework on a real test case, namely Java. We encode the Java Virtual Machine (JVM for short) operational semantics and Java bytecode programs into TRS and construct over-approximations of JVM states.

With regards to rewriting, the contribution of this paper is dual. First, we propose an encoding of a significant part of Java into *left-linear, unconditionnal* TRS. For rewriting, the second contribution is to have scaled up the theoretical construction of tree automata completion to the verification of Java bytecode

programs. With respect to static analysis, the contribution of this paper is to show that regular approximations can be used as a foundational mechanism for ensuring, by construction, safety of static analyzers. This paper shows that the approach can already be used to achieve standard class analysis on Java bytecode programs. Moreover, using approximation rules instead of abstract domains makes the analysis easier to fine-tune and to prove correct. This is of great interest, when a standard analysis is too coarse, since our technique permits to adapt the analysis to the property to prove and preserve safety.

The paper is organized as follows. Section 2 introduces the formal background of the rewriting theory. Section 3 shows how to over-approximate the set of reachable terms using tree automata. Section 4 presents a term rewriting model of the Java semantics. Section 5 presents, by the means of some classical examples, how rewriting approximations can be used for a class analysis. Section 6 compares our contribution with related works. Section 7 concludes.

2   Formal Background

Comprehensive surveys can be found in [6, 1] for term rewriting systems, and in [5, 14] for tree automata and tree language theory.

Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity function, and let $\mathcal{X}$ be a countable set of variables. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of terms, and $\mathcal{T}(\mathcal{F})$ denotes the set of ground terms (terms without variables). The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. A substitution is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$, which can be extended uniquely to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A position p for a term t is a word over $\mathbb{N}$. The empty sequence ϵ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by:

- $\mathcal{P}os(t) = \{\epsilon\}$ if $t \in \mathcal{X}$
- $\mathcal{P}os(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$

If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s. A term rewriting system $\mathcal{R}$ is a set of *rewrite rules* $l \rightarrow r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \rightarrow r$ is *left-linear* if each variable of l (resp. r) occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \rightarrow r$ of $\mathcal{R}$ is left-linear). The TRS $\mathcal{R}$ induces a rewriting relation $\rightarrow_{\mathcal{R}}$ on terms whose reflexive transitive closure is denoted by $\rightarrow_{\mathcal{R}}^{\star}$. The set of $\mathcal{R}$-descendants of a set of ground terms E is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \rightarrow_{\mathcal{R}}^{\star} t\}$.

The verification technique we propose in this paper is based on the computation of $\mathcal{R}^*(E)$. Note that $\mathcal{R}^*(E)$ is possibly infinite: $\mathcal{R}$ may not terminate and/or E may be infinite. The set $\mathcal{R}^*(E)$ is generally not computable [14]. However, it is possible to over-approximate it [8, 18] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. We next define tree automata.

Let $\mathcal{Q}$ be a finite set of symbols, with arity 0, called *states* such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*.

Definition 1 (Transition and normalized transition). *A* transition *is a rewrite rule* $c \to q$*, where c is a configuration i.e.* $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ *and* $q \in \mathcal{Q}$*. A* normalized transition *is a transition* $c \to q$ *where* $c = f(q_1, \ldots, q_n)$*,* $f \in \mathcal{F}$ *whose arity is n, and* $q_1, \ldots, q_n \in \mathcal{Q}$*.*

Definition 2 (Bottom-up non-deterministic finite tree automaton). *A bottom-up non-deterministic finite tree automaton (tree automaton for short) is a quadruple* $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$*, where* $\mathcal{Q}_f \subseteq \mathcal{Q}$ *and* Δ *is a set of normalized transitions.*

The *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the transitions of $\mathcal{A}$ (the set Δ) is denoted by $\to_\Delta$. When Δ is clear from the context, $\to_\Delta$ will also be denoted by $\to_\mathcal{A}$.

Definition 3 (Recognized language). *The tree language recognized by* $\mathcal{A}$ *in a state q is* $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_\mathcal{A}^\star q\}$*. The language recognized by* $\mathcal{A}$ *is* $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} \mathcal{L}(\mathcal{A}, q)$*. A tree language is regular if and only if it can be recognized by a tree automaton.*

3 Approximations of reachable terms

Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm, proposed in [10, 8], computes a tree automaton $\mathcal{A}_\mathcal{R}^k$ such that $\mathcal{L}(\mathcal{A}_\mathcal{R}^k) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when it is possible (for the classes of TRSs where an exact computation is possible, see [8]) and such that $\mathcal{L}(\mathcal{A}_\mathcal{R}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ otherwise.

The tree automata completion works as follows. From $\mathcal{A} = \mathcal{A}_\mathcal{R}^0$ completion builds a sequence $\mathcal{A}_\mathcal{R}^0 . \mathcal{A}_\mathcal{R}^1 \ldots \mathcal{A}_\mathcal{R}^k$ of automata such that if $s \in \mathcal{L}(\mathcal{A}_\mathcal{R}^i)$ and $s \to_\mathcal{R} t$ then $t \in \mathcal{L}(\mathcal{A}_\mathcal{R}^{i+1})$. If we find a fixpoint automaton $\mathcal{A}_\mathcal{R}^k$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_\mathcal{R}^k)) = \mathcal{L}(\mathcal{A}_\mathcal{R}^k)$, then we have $\mathcal{L}(\mathcal{A}_\mathcal{R}^k) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}_\mathcal{R}^0))$ (or $\mathcal{L}(\mathcal{A}_\mathcal{R}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ if $\mathcal{R}$ is not in one class of [8]). To build $\mathcal{A}_\mathcal{R}^{i+1}$ from $\mathcal{A}_\mathcal{R}^i$, we achieve a *completion step* which consists of finding *critical pairs* between $\to_\mathcal{R}$ and $\to_{\mathcal{A}_\mathcal{R}^i}$. To define the notion of critical pair, we extend the definition of substitutions to terms of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. For a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \to r \in \mathcal{R}$, a critical pair is an instance $l\sigma$ of l such that there exists $q \in \mathcal{Q}$ satisfying $l\sigma \to_{\mathcal{A}_\mathcal{R}^i}^* q$ and $l\sigma \to_\mathcal{R} r\sigma$. Note that since $\mathcal{R}$, $\mathcal{A}_\mathcal{R}^i$ and the set $\mathcal{Q}$ of states of $\mathcal{A}_\mathcal{R}^i$ are finite, there is only a finite number of critical pairs. For every critical pair detected between $\mathcal{R}$ and $\mathcal{A}_\mathcal{R}^i$ such that $r\sigma \not\to_{\mathcal{A}_\mathcal{R}^i}^* q$, the tree automaton $\mathcal{A}_\mathcal{R}^{i+1}$ is constructed by adding a new transition $r\sigma \to q$ to $\mathcal{A}_\mathcal{R}^i$ such that $\mathcal{A}_\mathcal{R}^{i+1}$ recognizes $r\sigma$ in q, i.e. $r\sigma \to_{\mathcal{A}_\mathcal{R}^{i+1}} q$.

$$
\begin{array}{ccc}
l\sigma & \xrightarrow{\ \ \mathcal{R}\ \ } & r\sigma \\
\mathcal{A}_\mathcal{R}^i \downarrow & {}^* \quad {}^* & \Big\downarrow \\
q & \dashleftarrow & \mathcal{A}_\mathcal{R}^{i+1}
\end{array}
$$
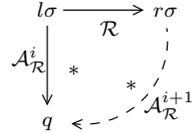
However, the transition $r\sigma \to q$ is not necessarily a normalized transition of the form $f(q_1, \ldots, q_n) \to q$ and so it has to be normalized first. Since normalization

consists in associating state symbols to subterms of the left-hand side of the new transition, it always succeed. Note that, when using new states to normalize the transitions, completion is as precise as possible. However, without approximation, completion is likely not to terminate (because of general undecidability results [14]). To enforce termination, and produce an over-approximation, the completion algorithm is parametrized by a set N of *approximation rules*. When the set N is used during completion to normalize transitions, the obtained tree automata are denoted by $\mathcal{A}_{N,\mathcal{R}}^1, \ldots, \mathcal{A}_{N,\mathcal{R}}^k$. Each such rule describes a context in which a list of rules can be used to normalize a term. For all $s, l_1, \ldots, l_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ and for all $x, x_1, \ldots, x_n \in \mathcal{Q} \cup \mathcal{X}$, the general form for an approximation rule is:

$$[s \rightarrow x] \rightarrow [l_1 \rightarrow x_1, \ldots, l_n \rightarrow x_n].$$

The expression $[s \rightarrow x]$ is a pattern to be matched with the new transitions $t \rightarrow q'$ obtained by completion. The expression $[l_1 \rightarrow x_1, \ldots, l_n \rightarrow x_n]$ is a set of rules used to normalize t. To normalize a transition of the form $t \rightarrow q'$, we match s with t and x with q', obtain a substitution σ from the matching and then we normalize t with the rewrite system $\{l_1\sigma \rightarrow x_1\sigma, \ldots, l_n\sigma \rightarrow x_n\sigma\}$. Furthermore, if $\forall i = 1 \ldots n : x_i \in \mathcal{Q}$ or $x_i \in \mathcal{V}ar(l_i) \cup \mathcal{V}ar(s) \cup \{x\}$ then since $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, $x_1\sigma, \ldots, x_n\sigma$ are necessarily states. If a transition cannot be fully normalized using approximation rules N, normalization is finished using some new states, see Example 1.

The main property of the tree automata completion algorithm is that, whatever the state labels used to normalize the new transitions, if completion terminates then it produces an over-approximation of reachable terms [8].

Theorem 1 ([8]). *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ be a tree automaton and N be a set of approximation rules. If completion terminates on $\mathcal{A}_{N,\mathcal{R}}^k$ then*

$$\mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$$

Here is a simple example illustrating completion and the use of approximation rules when the language $\mathcal{R}^*(E)$ is not regular.

Example 1. Let $\mathcal{R} = \{g(x, y) \rightarrow g(f(x), f(y))\}$ and let $\mathcal{A}$ be the tree automaton such that $\mathcal{Q}_f = \{q_f\}$ and $\Delta = \{a \rightarrow q_a, g(q_a, q_a) \rightarrow q_f\}$. Hence $\mathcal{L}(\mathcal{A}) = \{g(a, a)\}$ and $\mathcal{R}^*(E) = \{g(f^n(a), f^n(a)) \mid n \geq 0\}$. Let $N = [g(f(x), f(y)) \rightarrow z] \rightarrow [f(x) \rightarrow q_1, f(y) \rightarrow q_1]$. During the first completion step, we find a critical pair $g(q_a, q_a) \rightarrow_{\mathcal{R}} g(f(q_a), f(q_a))$ and $g(q_a, q_a) \rightarrow_{\mathcal{A}}^* q_f$. We thus have to add the transition $g(f(q_a), f(q_a)) \rightarrow q_f$ to $\mathcal{A}$. To normalize this transition, we match $g(f(x), f(y))$ with $g(f(q_a), f(q_a))$ and match z with q_f and obtain $\sigma = \{x \mapsto q_a, y \mapsto q_a, z \mapsto q_f\}$. Applying σ to $[f(x) \rightarrow q_1, f(y) \rightarrow q_1]$ gives $[f(q_a) \rightarrow q_1, f(q_a) \rightarrow q_1]$. This last system is used to normalize the transition $g(f(q_a), f(q_a)) \rightarrow q_f$ into the set $\{g(q_1, q_1) \rightarrow q_f, f(q_a) \rightarrow q_1\}$ which is added to $\mathcal{A}$ to obtain $\mathcal{A}_{N,\mathcal{R}}^1$. The completion process continues for another step and ends on $\mathcal{A}_{N,\mathcal{R}}^2$ whose set of transition is $\{a \rightarrow q_a, g(q_a, q_a) \rightarrow q_f, g(q_1, q_1) \rightarrow$

$q_f, f(q_a) \rightarrow q_1, f(q_1) \rightarrow q_1\}$. We have $\mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k) = \{g(f^n(a), f^m(a)) \mid n, m \geq 0\}$ which is an over-approximation of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.

The tree automata completion algorithm and the approximation mechanism are implemented in the Timbuk [13] tool. On the previous example, once the fixpoint automaton $\mathcal{A}_{N,\mathcal{R}}^k$ has been computed, it is possible to check whether some terms are reachable, i.e. recognized by $\mathcal{A}_{N,\mathcal{R}}^k$ or not. This can be done using tree automata intersections [8]. Another way to do that is to search instances for a pattern t, where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, in the tree automaton. Given t it is possible to check if there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a state $q \in \mathcal{Q}$ such that $t\sigma \rightarrow_{\mathcal{A}_{N,\mathcal{R}}^k}^* q$. If such a solution exists then it proves that there exists a term $s \in \mathcal{T}(\mathcal{F})$, a position $p \in \mathcal{P}os(s)$ and a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $s[t\sigma']_p \in \mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e. that $t\sigma'$ occurs as a subterm in the language recognized by $\mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k)$. On the other hand, if there is no solution then it proves that no such term is in the over-approximation, hence it is not in $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e. it is not reachable.

In the patterns we use in this paper, '$?x$' denotes variables for which a value is wanted and '$\_$' denotes anonymous variables for which no value is needed. For instance, the pattern $g(f(\_), g(\_, \_))$ has no solution on $\mathcal{A}_{N,\mathcal{R}}^2$ of example 1, meaning that no term containing any ground instance of this pattern is reachable from $g(a, a)$.

4   Formalization of the Java Bytecode Semantics using Rewriting Rules

This section describes how to formalize the semantics of an object-oriented language (here, Java bytecode) using rewriting rules. From a bytecode Java program p, we have developed a prototype that automatically produces a TRS $\mathcal{R}$ modeling a significant part of the Java semantics (stacks, frames, objects, references, methods, heaps, integers) as well as the semantics of p. For the moment, exceptions and threads are not taken into account but they can be elegantly encoded using rewriting [16, 7]. The formalization follows the structure of standard Java semantics formalizations [2, 9].

4.1   Formalization of Java Program States

A Java program state contains a current execution frame (also called activation record), a frame stack, a heap, and a static heap. A frame gives information about the method currently being executed: its name, current program counter, operand stack and local variables. When a method is invoked the current frame is stored in the frame stack and a new current frame is created. A heap is used to store objects and arrays, i.e. all the information that is not local to the execution of a method. The static heap stores values of static fields, i.e. values that are shared by all objects of a same class.

Let P be the infinite set of all the possible Java programs. Given $p \in P$, let $C(p)$ be the corresponding finite set of class identifiers and $C_r(p)$ be $C(p) \cup \{array\}$. A value is either a primitive type or a reference pointing to an object (or an array) in the heap. In our setting, we only consider integer and boolean primitive types. Let $PC(p)$ be the set of integers from 0 to the highest possible program point in all the methods in p. Let $M(p)$ be the set of method names and $M_{id}(p)$ be the finite set of pairs (m, c) where $m \in M(p)$, $c \in C(p)$ and m is a method defined by the class c. This last set is needed to distinguish between methods having the same name but defined by different classes. For the sake of simplicity, we do not distinguish between methods having the same name but a different signature but this could easily be done.

Following standard Java semantics we define a *frame* to be a tuple $f = (pc, m, s, l)$ where $pc \in PC(p)$, $m \in M_{id}(p)$, s an operand stack, l a finite map from indexes to values (local variables). An object from a class c is a map from field identifiers to values. The heap is a map from references to objects and arrays. The static heap is a map from static field names to values. A program state is a tuple $s = (f, fs, h, k)$ where f is a frame, fs is a stack of frames, h is a heap and k a static heap.

4.2 A Program State as a Term

Let $\mathcal{F}_C(p) = C(p)$ and $\mathcal{F}_{C_r}(p) = C_r(p) = C(p) \cup \{array\}$ be sets of symbols. We encode a reference as a term $loc(c, a)$ where $c \in C_r(P)$ is the class of the object being referenced and a is an integer. This is coherent with Java semantics where it is always possible to know dynamically the class of an object corresponding to a reference. We use $\mathcal{F}_{primitive} = \{succ : 1, pred : 1, zero : 0\}$ primitive types (integers), $\mathcal{F}_{reference}(p) = \{loc : 2, succ : 1, zero : 0\} \cup \mathcal{F}_{C_r}(p)$ for references and $\mathcal{F}_{value}(p) = \mathcal{F}_{primitive} \cup \mathcal{F}_{reference}(p)$ for values. For example, $loc(foo, succ(zero))$ is a reference pointing to the object located at the index 1 in the foo class heap. Let x be the higher program point of the program (p), then $\mathcal{F}_{PC}(p) = \{pp0 : 0, pp1 : 0, ..., ppx : 0\}$. $F_M(p)$ is defined the same way as $F_C(p)$. $\mathcal{F}_{M_{id}}(p) = \{name : 2\} \cup \mathcal{F}_M(p) \cup \mathcal{F}_C(p)$. For example $name(bar, A)$ stands for the method bar defined by the class A. Let $l(p)$ denote the maximum of local variables used by the methods of the program package p. We use $\mathcal{F}_{stack}(p) = \{stack : 2, nilstack : 0\} \cup \mathcal{F}_{value}(p)$ for stacks, $\mathcal{F}_{localVars}(p) = \{locals : l(p), nillocal : 0\} \cup \mathcal{F}_{value}(p)$ for local variables and $\mathcal{F}_{frame}(p) = \{frame : 4\} \cup \mathcal{F}_{PC}(p) \cup \mathcal{F}_{M_{id}}(p) \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{localVars}(p)$ for frames. A possible frame thus would be: $frame(name(bar, A), pp4, stack(succ(zero), nilstack), locals(loc(bar, zero), nillocal))$ where the program counter points to the 4th instruction of the method bar defined by the class A. The current operand stack just has the integer 1 on the top. The first local variable is some reference and the other is not initialized.

The alphabet $\mathcal{F}_{objects}(p)$ contains the same symbols as $\mathcal{F}_C(p)$, where the arity of each symbol is the corresponding number of non-static fields. As an example, $objectA(zero)$ is an object from the class A with one field whose value is $zero$.

Let nc be the number of classes. We chose to divide the heap into nc *class heaps* plus one for the arrays. In a reference $loc(c, a)$, a is the index of the

object in the list representing the heap of class c. An array is encoded using a list and indexes in a similar way. We use $\mathcal{F}_{heap}(p) = \{heaps : (nc + 1), heap : 2\} \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{objects}(p)$ for heaps, and $\mathcal{F}_{state}(p) = \{state : 4\} \cup \mathcal{F}_{frame}(p) \cup \mathcal{F}_{heap}(p)$ for states.

4.3   Java Bytecode Semantics

In this section we will show some representative examples of the Java Bytecode Semantics.

$$(pop) \; \frac{(m, pc, x :: s, l)}{(m, pc + 1, s, l)} \qquad (store_i) \; \frac{(m, pc, x :: s, l)}{(m, pc + 1, s, x \rightarrow_i l)}$$

Fig. 1. Example of bytecodes operating at the frame level

Figure 1 presents some rules of the semantics operating at the frame level. For a given instruction, if a frame matches the top expression then it is transformed into the lower expression. Considering the frame (pc, m, s, l), pc denotes the current program point, m the current method identifier, s the current stack and l the current array of local variables. The operator '::' models stack concatenation. The $store_i$ instruction is used to store the value at the top of the current stack in the ith register, where $x \rightarrow_i l$ denotes the new resulting array of local variables.

$$\frac{(invokeVirtual_{name})}{((m, pc, ref :: s, l), fs, h, k), c = class(ref, h, k), m' = lookup(name, c)}{((m', 0, [], storeparams(ref :: s, m')), (m, pc + 1, popparams(ref :: s, m'), l) :: fs, h, k)}$$

Fig. 2. Example of bytecodes operating at the state level

Figure 2 presents a rule of the semantics operating at the state level. For a state $((m, pc, s, l), fs, h, k)$, the symbols pc, m, s and l denote the current frame components, fs the current stack of frames, h the heap and k the static heap. The instruction $invokeVirtual_{name}$ implements dynamic method invocation. The method to be invoked is determined from its *name* and the class of the reference at the top of the stack. The internal function $class(ref, h, k)$ is used to get the reference's class c and $lookup(name, c)$ searches the class hierarchy in a bottom-up fashion for the the method m' corresponding to this name and this class. We use two internal functions to manage the parameters of the method (pushed on the stack before invoking): $storeparams(ref :: s, m')$ to build an array of local variables from values on the top of the operand stack and $popparams(ref :: s, m')$ to remove from the current operand stack the parameters used by m'. With those tools, it is possible to build a new frame pointing at

the first program point of m' and to push the current frame on the frame stack. Some other examples can be found in [3].

4.4   Java Bytecode Semantics using Rewriting Rules

In this section we analyze in more detail how to encode an operational semantics of a programming language into rewriting rules in a way that makes the resulting system amenable to approximation by the techniques described in this paper. The first constraint is that the term rewriting system has to be left-linear (see Theorem 1). The second constraint, is that intermediate steps modeling internal functions (such as $popparams$, $storeparams$, ...), should not appear in the final result and must thus be easy to filter out. To this end, we introduce a notion of intermediate frames (named $xframe$). We can express the Java Bytecode Semantics of a Java bytecode program p by means of rewriting rules (see [3] details). We give here the encodings of pop and $invokeVirtual$ instructions.

In the following, symbols m, c, pc, s, l, fs, h, k, x, y, a, b, adr, $l0$, $l1$, $l2$, $size$, h, $h0$, $h1$, ha are variables. For a given program point pc in a given method m, we build an $xframe$ term very similar to the original $frame$ term but with the current instruction explicitly stated. The $xframe$s are used to compute intermediate steps. If an instruction requires several internal rewriting steps, we will only rewrite the corresponding $xframe$ term until the execution of the instruction ends. Assume that, in program p, the instruction at program point $pp2$ of method foo of class A is pop. Rule 1 builds a $xframe$ term by explicitly adding the current instruction to the $frame$ term. Rule 2 describes the semantics of pop. Rule 3 specifies the control flow by defining the next program point.

1	$frame(name(foo,A),pp2,s,l)$	$\rightarrow xframe(pop,name(foo,A),pp2,s,l)$
2	$xframe(pop,m,pc,stack(x,s),l)$	$\rightarrow frame(m,next(pc),s,l)$
3	$next(pp2)$	$\rightarrow pp3$

Fig. 3. An example pop instruction by rewriting rules, for program p

Now, assume that, in program p, the instruction at program point $pp2$ of method foo of class A is $invokeVirtual$. This instruction requires to compile some information about methods and the class hierarchy into the rules. Basically, we need to know what is the precise method to invoke, given a class identifier and a method name. In p, assume that A and B are two classes such that B extends A. Let set be a method implemented in the class A (and thus available from B) with one parameter and $reset$ a method implemented in the class B (and thus unavailable from A) with no parameter. Figure 4 presents the resulting rules for this simple example.

To complete the modeling of the semantics and the program by rewriting rules we need stubs for native libraries used by the program. At present, we have developed stubs for some of the methods from the $javaioInputStream$

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow xframe(invokeVirtual(set),$
		$name(foo, A), pp2, s, l)$
2	$state(xframe(invokeVirtual(set),$	$\rightarrow state(frame(name(set, A), pp0, s,$
	$m, pc, stack(loc(A, adr),$	$locals(loc(A, adr), x, nillocal)),$
	$stack(x, s)), l), fs, h, k)$	$stack(storedframe(m, pc, s, l), fs), h, k)$
3	$state(xframe(invokeVirtual(set),$	$\rightarrow state(frame(name(set, A), pp0, s,$
	$m, pc, stack(loc(B, adr),$	$locals(loc(B, adr), x, nillocal)),$
	$stack(x, s)), l), fs, h, k)$	$stack(storedframe(m, pc, s, l), fs), h, k)$
4	$state(xframe(invokeVirtual(reset),$	$\rightarrow state(frame(name(reset, B), pp0, s,$
	$m, pc, stack(loc(B, adr), s), l), fs, h, k)$	$locals(loc(B, adr), nillocal, nillocal)),$
		$stack(storedframe(m, pc, s, l), fs), h, k)$
5	$next(pp2)$	$\rightarrow pp3$

Fig. 4. $invokeVirtual_{set}$ instruction by rewriting rules

class. We model interactions of a Java program state with its environment using a term of the form $IO(s, i, o)$ where s is the state, i is the input stream and o the output stream.

5 Class Analysis as a Rewriting Theory

In most program analyzes, it is often necessary to know the control flow graph. For Java, as for other object-oriented languages, the control flow depends on the data flow. When a method is invoked, to know which one is executed, the class of the involved object is needed. For instance, on the Java program of Figure 5, `x.foo()` calls `this.bar()`. To know which version of the `bar` is called, it is necessary to know the class of `this` and thus the class of `x` in `x.foo()` call. The method actually invoked is determined dynamically during the program run. Class analysis aims at statically determining the class of objects stored in fields and local variables, and allows to build a more precise control flow graph valid for all possible executions. Note that in this example, exceptions around `System.in.read()` are required by the Java compiler. However, in this paper, we do not take them into account in the control flow.

There are different standard class analyzes, from simple and fast to precise and expensive. We consider k-CFA analysis [17]. In these analyzes, primitive types are abstracted by the name of their type and references are abstracted by the class of the objects they point to. In 0-CFA analysis, each method is analyzed only once, without distinguishing between the different calls (and hence the arguments passed) to this method. k-CFA analyzes different calls to the same method separately, taking into account up to k frames on the top of the frame stack.

Starting from a term rewriting system $\mathcal{R}$ modeling the semantics of a Java program, and a tree automaton $\mathcal{A}$ recognizing a set of initial Java program states, we aim at computing an automaton $\mathcal{A}_{N,\mathcal{R}}^k$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. We developed a prototype which produces $\mathcal{R}$ and $\mathcal{A}$ from a Java `.class` file.

```
class A{                                    o1= new A();
    int y;                                  o2= new B();
    void foo(){this.bar();}                 try{
    void bar(){y=1;}                          x=System.in.read();
}                                           }
class B extends A{                          catch (java.io.IOException e)
    void bar(){y=2;}                          { x = 0;}
}                                         while (x != 0){
class Test{                                 execute(o1);
    public void execute(A x){               execute(o2);
        x.foo();                            try{
    }                                         x=System.in.read(); }
    public void main(String[] argv){        catch (java.io.IOException e)
        A o1;                                 { x = 0;}}
        B o2;                             }
        int x;                            }
```

Fig. 5. Java Program Example

From the Java source program of Figure 5, one can obtain the files `Test.class`, `A.class` and `B.class` whose content is around 90 lines of bytecode. The TRS $\mathcal{R}$ produced by compilation of those classes is composed of 275 rewrite rules. The number of rewrite rules is linear w.r.t. the size of the bytecode files. The analysis itself is performed using Timbuk [13]. Successively, this section details a 0-CFA, a 1-CFA and an even more precise analysis obtained using the same TRS $\mathcal{R}$ and automaton $\mathcal{A}$, but using different sets of approximation rules. On this program, the set of reachable program states is infinite (and thus approximations are necessary) because the instruction `x=System.in.read()`, reading values in the input stream, is embedded in an unbounded loop. As long as the value stored in the variable `x` is different from `0`, the computation continues. Moreover, since we want to analyze this program for any possible stream of integers, in the automaton $\mathcal{A}$ the input stream is unbounded.

5.1 0-CFA Analysis

For a 0-CFA analysis, all integers are abstracted by their type, i.e. they are defined by the following transitions in $\mathcal{A}$: $zero \rightarrow q_{int}$, $succ(q_{int}) \rightarrow q_{int}$ and $pred(q_{int}) \rightarrow q_{int}$. The input stream is also specified by $\mathcal{A}$ as an infinite stack of integers: $nilstackin \rightarrow q_{in}$ and $stackin(q_{int}, q_{in}) \rightarrow q_{in}$. Approximation rules for integers, streams and references are defined by: $[x \rightarrow y] \rightarrow [zero \rightarrow q_{int}, succ(q_{int}) \rightarrow q_{int}, pred(q_{int}) \rightarrow q_{int}, nilstackin \rightarrow q_{in}, stackin(q_{int}, q_{in}) \rightarrow q_{in}, loc(A, \alpha) \rightarrow q_{refA}, loc(B, \beta) \rightarrow q_{refB}]$ where x, y, α and β are variables. The pattern $[x \rightarrow y]$ matches any new transition to normalize and the rules $loc(A, x) \rightarrow q_{refA}$ and $loc(B, y) \rightarrow q_{refB}$ merge all references to an object of the class A and an object of the class B into the states q_{refA} and q_{refB}, respectively.
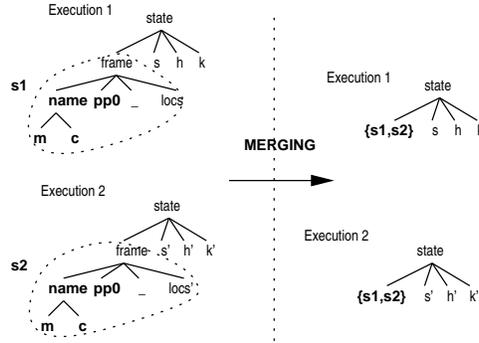
Fig. 6. Principle of approximation rules for a 0-CFA analysis

The approximation rules for frames and states are built according to the principle illustrated in Figure 6. The frames representing two different calls to the method m of the class c are merged independently of the current state of the execution in which the method m is called. The set of approximation rules N is completed by giving such an approximation rule for each method of each class. Using N, we can automatically obtain a fixpoint automaton $\mathcal{A}_{N,\mathcal{R}}^{145}$ over-approximating the set of all reachable Java program states. The result of the 0-CFA class analysis can be obtained, for each program location (a program point in a method in a class), by asking for the possible classes for each object in the stack or in the local variables. For instance, to obtain the set of possible classes $?c$ for the object passed as parameter to the method *execute*, i.e. the possible classes for the second local variable at program point $pp0$ of *execute*, one can use the following pattern: $frame(name(execute, Test), pp0, \_, locals(\_, loc(?c, \_), ...))$. The result obtained for this pattern is that there exist two possible values for $?c$: q_A and q_B which are the states recognizing respectively the classes A and B. This is consistent with 0-CFA which is not able to discriminate between the two possible calls to the *execute* method.

5.2   1-CFA Analysis

For 1-CFA, we need to refine the set of approximation rules into N'. In N' the rules on integers, the input stream and references are similar to the ones used for 0-CFA. In N', approximation rules for states and frames are designed according to the principle illustrated in Figure 7. Contrary to Figure 6, the frames for the method m of the class c are merged if the corresponding method calls have been done from the same program point (in the same method m' of the class c'). For example, there are two approximation rules for the method *execute* of the class $Test$: one applying when *execute* is invoked from the program point $pp18$ of the method *main*, and one applying when it is done from the program point $pp21$ of this same method. Applying the same principle for all the methods, we obtain
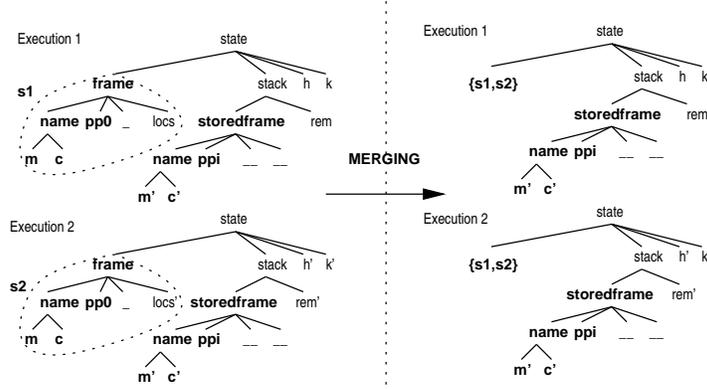
Fig. 7. Principle of approximation rules for a 1-CFA analysis

a complete set of approximation rules N'. Using N', completion terminates on $\mathcal{A}_{N',\mathcal{R}}^{140}$. The following patterns:

$$state(frame(name(execute, Test), pp0, \_, locals(\_, loc(?c, \_), ...)), stack(storeframe(\_, pp18, ...), \_)...)$$

$$state(frame(name(execute, Test), pp0, \_, locals(\_, loc(?c, \_), ...)), stack(storeframe(\_, pp21, ...), \_)...)$$

gives the desired result: each pattern has only one solution for $?c$: q_A for the first and q_B for the second. Using a similar pattern to query the 0-CFA automaton $\mathcal{A}_{N,\mathcal{R}}^{145}$, gives q_A and q_B as solution for $?c$ for both program points.

5.3 Fine-tuning the precision of the analysis

Assume that we want to show that, after the execution of the previous program, field y has always a value 1 for objects of class A and 2 for objects of class B. This cannot be done by 1-CFA nor by any k-CFA since, in those analyzes, integers are abstracted by their type. One of the advantage of our technique is its ability to easily make approximation more precise by removing some approximation rules.

The property we want to prove is related to values 1 and 2 so it is tempting to refine our approximation so as not to merge those values. However, only distinguishing these two values is not enough for the analysis to succeed. Further experimentation with the approximation shows that refining the approximation of the integers by distinguishing between 0, 1, 2 and "any other integer" is enough to prove the desired property. Formally, this is expressed by the following transitions: $0 \to q_0, succ(q_0) \to q_1, succ(q_1) \to q_2, succ(q_2) \to q_{int}, succ(q_{int}) \to q_{int}$. For specifying the negative integers, the following transitions are used: $pred(q_0) \to q_{negint}$ and $pred(q_{negint}) \to q_{negint}$. The input stream representation is also modified by the following transitions: $nilstackin \to q_{in}$,

$stackin(q_{negint}, q_{in}) \rightarrow q_{in}$, $stackin(q_{int}, q_{in}) \rightarrow q_{in}$ and $stackin(q_j, q_{in}) \rightarrow q_{in}$ with $j = 0, \ldots, 2$.

No other approximation is needed to ensure termination of the completion. In the fixpoint automaton $\mathcal{A}_{N,\mathcal{R}}^{161}$, we are then able to show that, when the Java program terminates, there is only two possible configurations of the heap. Either the heap contains an object of class A and an object of class B whose fields are both initialized to 0, or it contains an object of class A whose field has the value 1 and an object of class B whose field has the value 2. These verifications have been performed using a pattern matching with all the frames whose pp value is the last control point of the program.

This result is not surprising. The first result is possible when there is zero iterations of the loop (x is set to 0 before the instruction `while (x != 0){...}`). The second result is obtained for 1 or more iterations. Nevertheless, this kind of result is impossible to obtain with the two previous analyzes presented in Section 5.1 and 5.2.

6   Related work

Term rewriting systems have been used to define and prototype semantics for a long time. However, this subject has recently reappeared for verification purposes. In [7, 16], rewriting is also used as operational semantics for Java. The verification done on the obtained rewriting system is closer to finite model-checking or to simulation, since it can only deal with finite state programs. Moreover, no abstraction mechanism is proposed. Hence, our work is complementary to theirs since it permits to define abstractions in the rewrite model and to prove properties on Java applets for unbounded sets of inputs or for unbounded execution paths. In [15], abstractions on reachability analysis are defined but they seem to be too restrictive to deal with programming language semantics. Instead of tree automata, Meseguer, Palomino and Martí-Oliet use equations to define approximated equivalence classes. More precisely, they use terminating and confluent term rewriting systems normalizing every term of a class to its representative. In order to guarantee safety of approximations, approximation and specification rules must satisfy strong syntactic constraints. Roughly, approximation TRS and specification TRS, they use, have to commute. Such properties are hard to prove on a TRS encoding the Java semantics. Moreover, the approximation rules we used for class analysis are contextual and cannot easily be expressed as equations.

Takai [18] also proposed a theoretical version of approximated reachability analysis over term rewriting systems. This work also combines equations and tree automata. However, again, syntactic restrictions imposed on the equations are strong and would prevent us from constructing the kind of approximation we use on Java bytecode.

In [4], Bouajjani et al. propose a verification methodology based on abstractions and tree transducer applications on tree automata languages, called *Abstract Regular Tree Model Checking*. This brings into play a tree transducer τ, a

tree automaton $\mathcal{A}$ and an abstraction α. For a given system to verify, τ encodes its transition relation and $\mathcal{L}(\mathcal{A})$ accounts for its set of initial configurations. As for computing $\mathcal{R}^*$ of a set of terms in rewriting, computing $\tau^*(\mathcal{L}(\mathcal{A}))$ may not terminate. A well-suited abstraction α makes the computation converge at the expense of an over-approximation of the set of configurations actually reachable. The underlying idea of this technique is close to ours. However, in our case, the TRS can implement basic computations in the semantics which would be complicated to specify in terms of transducers.

As in classical static class analyzes (such as *e.g.*, [17]), we can get several ranges of precision of k-CFA, depending of the approximation rules. In addition, starting from an automatically generated approximation, it is possible to adapt approximation rules so as to get a more precise abstraction and prove specific properties that may be difficult to show by an analyzer whose abstractions are built-in (See Section 5.3 for instance).

7   Conclusion

We have defined a technique, based on rewriting and tree automata, for prototyping static analyzers from the operational semantics of a programming language. As a test case, we showed how to produce a TRS $\mathcal{R}$ modeling the operational semantics of a given Java program p. In this setting, given a set of inputs E the set $\mathcal{R}^*(E)$ represents the set of program states reachable by p on inputs E, i.e. the collecting semantics of p. The TRS $\mathcal{R}$ is produced automatically and has a size linear in the size of the source program p. The technique has been implemented and experimented on a number of standard control-flow analyzes for Java bytecode, demonstrating the feasibility of the technique.

The exact set of reachable states is not computable in general so we use the tree automata completion algorithm and approximation rules so as to compute a finite approximation of the superset of $\mathcal{R}^*(E)$. The approximation technique works at the level of terms and their representation through a tree automata and has a number of advantages. First, the correctness of the approximation is guaranteed by the underlying theory and does not have to be proved for each proposed abstraction. Second, the analysis has easy access to several types of information (on integers, memory, call stacks), as illustrated in Section 5.3. This is an advantage compared to the more standard approach which combines several data flow analyzes (by techniques such as reduced and open product) to gather the same information. Third, it is relatively easy to fine-tune the analysis by adding and removing approximation rules.

We have presented our approach in terms of a sequential fragment of Java bytecode but the term rewriting setting is well suited to deal with the extension to concurrent aspects of Java and to the handling of exceptions [16, 7].

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

2. P. Bertelsen. Semantics of Java Byte Code. Technical report, Technical University of Denmark, 1997.

3. Y. Boichut, T. Genet, T. Jensen, and L. Le Roux. Rewriting Approximations for Fast Prototyping of Static Analyzers. Research Report RR 5997, INRIA, 2006. http://www.irisa.fr/lande/genet/publications.html.

4. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In *Proceedings of 7th International Workshop on Verification of Infinite-State Systems – INFINITY 2005*, number 4 in BRICS Notes Series, pages 15–24, 2005.

5. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata/, 2002.

6. N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

7. A. Farzan, C. Chen, J. Meseguer, and G. Rosu. Formal analysis of java programs in javafan. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

8. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33 (3-4):341–383, 2004.

9. Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. *ACM SIGPLAN Notices*, 34(10):147–166, 1999.

10. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proc. 9th RTA Conf., Tsukuba (Japan)*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.

11. T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proc. 17th CADE Conf., Pittsburgh (Pen., USA)*, volume 1831 of *LNAI*. Springer-Verlag, 2000.

12. T. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In *In Proceedings of Workshop on Issues in the Theory of Security*, 2003.

13. T. Genet and V. Viet Triem Tong. Timbuk 2.0 – a Tree Automata Library. IRISA / Université de Rennes 1, 2000. http://www.irisa.fr/lande/genet/timbuk/.

14. R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.

15. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational Abstractions. In *Proc. 19th CADE Conf., Miami Beach (Fl., USA)*, volume 2741 of *LNCS*, pages 2–16. Springer, 2003.

16. J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In *IJCAR*, pages 1–44, 2004.

17. O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991.

18. T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *Proc. 15th RTA Conf., Aachen (Germany)*, volume 3091 of *LNCS*, pages 119–133. Springer, 2004.