INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Rewriting Approximations for Fast Prototyping of Static Analyzers

Yohan Boichut , Thomas Genet , Thomas Jensen , Luka Leroux

N°5997

13 Octobre 2006

Systèmes communicants — Systèmes cognitifs

Rapport de recherche

ISSN 0249-6399

Rewriting Approximations for Fast Prototyping of Static Analyzers

Yohan Boichut , Thomas Genet , Thomas Jensen , Luka Leroux

Systèmes communicants — Systèmes cognitifs
Projet Lande

Rapport de recherche n° 5997 — 13 Octobre 2006 — 20 pages

Abstract: This paper defines a new framework for fast prototyping of static analyzers based on 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 program states that can be reached. With this approximation, it is possible to prove a variety of safety or security properties expressible in terms of (un)reachability. Compared with static analysis based on abstract interpretation, a salient feature of this approach is that it is correct by construction. The approach enables fast prototyping of static analyzers because modifying the analysis simply amounts to changing the set of rewrite rules defining the approximation. To illustrate the framework proposed here on a realistic programming language we instantiate it with the Java Virtual Machine semantics and use Java bytecode programs as running examples. We show how to compile a Java bytecode program into an equivalent term rewriting system and show how to quickly specify and implement a class analysis by defining rewriting approximations.

Key-words: Term Rewriting Systems, Static Analysis, Reachability, Approximation, Java, Bytecode, Tree Automata

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Approximations de réécriture pour le prototypage rapide d'analyseurs statiques

Résumé : Cet article définit un nouveau cadre pour le prototypage rapide d'analyseurs statiques basé sur des techniques de réécriture. A partir d'un système de réécriture représentant la sémantique opérationnelle du langage de programmation cible et d'un programme à analyser, nous construisons automatiquement une sur-approximation de l'ensemble des termes atteignables, c-à-d des états de programme atteignables. A l'aide de ces approximations, il est possible de prouver une grande variété de propriétés de sûreté et de sécurité exprimées sous la forme d'un problème de (non) atteignabilité. Comparée à l'analyse statique basée sur l'interprétation abstraite, une caractéristique intéressante de cette approche est qu'elle est correcte par construction. Cette approche permet de prototyper rapidement des analyseurs statiques car pour modifier le type d'analyse effectué, il est suffisant de modifer l'ensemble de règles de réécriture définissant l'approximation. Afin d'illustrer cette méthode de vérification sur un langage de programmation réaliste, nous avons choisi de l'instancier avec la sémantique de la machine virtuelle de Java (JVM) et d'analyser des programmes Java bytecode. Nous montrons comment compiler un programme Java en un système de réécriture équivalent et nous montrons également comment rapidement spécifier et implémenter des analyses de classe simples en définissant des règles d'approximation.

Mots-clé : Systèmes de réécriture, Analyse statique, Atteignabilité, Approximation, Java, Bytecode, Automates d'arbres

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 prototyping 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, 7, 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:

- an encoding of the operational semantics of the language as a term rewriting system (TRS for short),
- a translation scheme for transforming programs into rewrite rules,
- and an over-approximation of the set of reachable program states represented by a tree automaton, based on the tree automata completion algorithm [7].

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-approximation of JVM states.

With regards to rewriting, the main contribution of this paper is to have scaled up a theoretical construction, namely tree automata completion, to the verification of Java bytecode programs. With regards 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 is a first step in that direction and 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 analysis easier to prototype and to tune. 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 mean of some classical examples, how rewriting approximations can be used for a class analysis. Section 6 compares with related works. Section 7 concludes.

2 Formal Background

Comprehensive surveys can be found in [5, 2] for term rewriting systems, in [4, 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 uniquely be extended 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* $l \to r$, 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 $l \to r$ 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 $l \to r$ of $\mathcal{R}$ is left-linear). The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms whose reflexive transitive closure is denoted by $\to^{\star}_{\mathcal{R}}$. The set of $\mathcal{R}$-descendants of a set of ground terms E is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \to^{\star}_{\mathcal{R}} t\}$.

Example 1 *Let $\mathcal{R}$ be the TRS such that $\mathcal{R} = \{f(x) \to g(f(x))\}$. The term $f(a)$ rewrites to $g(f(a))$, i.e. $f(a) \to_{\mathcal{R}} g(f(a))$. Similarly, $g(f(a)) \to_{\mathcal{R}} g(g(f(a)))$. We thus have $f(a) \to_{\mathcal{R}}{}^{*} g(g(f(a)))$. Let E be the set of terms $E = \{f(a)\}$. On this example, we have $\mathcal{R}^*(E) = \{f(a), g(f(a)), g(g(f(a))), \ldots\} = \{g^*(f(a))\}$.*

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 (like in the previous example): $\mathcal{R}$ 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 [7, 19] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. We next define tree automata.

Let $\mathcal{Q}$ be an infinite 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 $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to^{\star}_{\mathcal{A}} q\}$. 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.*

Example 2 *Let $\mathcal{A}$ be the tree automaton $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ such that $\mathcal{F} = \{f, g, a\}$, $\mathcal{Q} = \{q_0, q_1\}$, $\mathcal{Q}_f = \{q_0\}$ and $\Delta = \{f(q_0) \to q_0, g(q_1) \to q_0, a \to q_1\}$. In Δ transitions are* normalized. *A transition of the form $f(g(q_1)) \to q_0$ is not normalized. The term $g(a)$ is a term of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ (and of $\mathcal{T}(\mathcal{F})$) and can be rewritten by Δ in the following way: $g(a) \to_\Delta g(q_1) \to_\Delta q_0$. Note that $\mathcal{L}(\mathcal{A}, q_1) = \{a\}$ and $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}, q_0) = \{g(a), f(g(a)), f(f(g(a))), \ldots\} = \{f^\star(g(a))\}$.*

3 Approximations of reachable terms

Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm, proposed in [10, 7], computes a tree automaton $\mathcal{A}^k_\mathcal{R}$ such that $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when it is possible (for the classes of TRSs where an exact computation is possible, see [7]) and such that $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ otherwise.

The tree automata completion works as follows. From $\mathcal{A} = \mathcal{A}^0_\mathcal{R}$ completion builds a sequence $\mathcal{A}^0_\mathcal{R}.\mathcal{A}^1_\mathcal{R} \ldots \mathcal{A}^k_\mathcal{R}$ of automata such that if $s \in \mathcal{L}(\mathcal{A}^i_\mathcal{R})$ and $s \to_\mathcal{R} t$ then $t \in \mathcal{L}(\mathcal{A}^{i+1}_\mathcal{R})$. If we find a fixpoint automaton $\mathcal{A}^k_\mathcal{R}$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A}^k_\mathcal{R})) = \mathcal{L}(\mathcal{A}^k_\mathcal{R})$, then we have $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}^0_\mathcal{R}))$ (resp. $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ if $\mathcal{R}$ is not in one class of [7]). To build $\mathcal{A}^{i+1}_\mathcal{R}$ from $\mathcal{A}^i_\mathcal{R}$, we achieve a *completion step* which consists in finding *critical pairs* between $\to_\mathcal{R}$ and $\to_{\mathcal{A}^i_\mathcal{R}}$. 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 $l\sigma \to^*_{\mathcal{A}^i_\mathcal{R}} q$ and $l\sigma \to_\mathcal{R} r\sigma$. Note that since $\mathcal{R}$, $\mathcal{A}^i_\mathcal{R}$ and the set $\mathcal{Q}$ of states of $\mathcal{A}^i_\mathcal{R}$ are finite, there is only a finite number of critical pairs. For every critical pair detected between $\mathcal{R}$ and $\mathcal{A}^i_\mathcal{R}$ such that $r\sigma \not\to^*_{\mathcal{A}^i_\mathcal{R}} q$, the tree automaton $\mathcal{A}^{i+1}_\mathcal{R}$ is constructed by adding a new transition $r\sigma \to q$ to $\mathcal{A}^i_\mathcal{R}$ such that $\mathcal{A}^{i+1}_\mathcal{R}$ recognizes $r\sigma$ in q, i.e. $r\sigma \to_{\mathcal{A}^{i+1}_\mathcal{R}} q$.

$$\begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_\mathcal{R}"', "*"] & r\sigma \arrow[dl, dashed, bend left, "*"', "\mathcal{A}^{i+1}_\mathcal{R}"] \\ q & \end{tikzcd}$$

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. For example, to normalize a transition of the form $f(g(a), h(q')) \to q$, we need to find some states q_1, q_2, q_3 and replace the previous transition by a set of normalized transitions: $\{a \to q_1, g(q_1) \to q_2, h(q') \to q_3, f(q_2, q_3) \to q\}$.

If q_1, q_2, q_3 are new states, then adding the transition itself or its normalized form does not make any difference. On the opposite, if we identify q_1 with q_2, the normalized form becomes $\{a \to q_1, g(q_1) \to q_1, h(q') \to q_3, f(q_1, q_3) \to q\}$. This set of normalized transitions represents the regular set of non-normalized transitions of the form $f(g^\star(a), h(q')) \to q$ which contains the transition we want to add but also many others. Hence, this is an over-approximation. We could have made an even more drastic approximation by identifying q_1, q_2, q_3 with q, for instance.

When always 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 $\mathcal{A}^1_{N,\mathcal{R}}, \ldots, \mathcal{A}^k_{N,\mathcal{R}}$. 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:

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

. The expression $[s \to x]$ is a pattern to be matched with the new transitions $t \to q'$ obtained by completion. The expression $[l_1 \to x_1, \ldots, l_n \to x_n]$ is a set of rules used to normalize t. To normalize a transition of the form $t \to q'$, 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 $\{l_1\sigma \to x_1\sigma, \ldots, l_n\sigma \to x_n\sigma\}$. 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 $x_1\sigma, \ldots, x_n\sigma$ are necessarily states. If a transition cannot be fully normalized using approximation rules N, normalization is finished using some new states.

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 [7]. In other words, approximation safety does not depend on the set of approximation rules used. Since the role of approximation rules is only to select particular states for normalizing transitions, the safety theorem of [7] can be reformulated in the following way.

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

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

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

Example 3 *Let $\mathcal{R} = \{g(x,y) \to g(f(x), f(y))\}$ and let $\mathcal{A}$ be the tree automaton such that $\mathcal{Q}_f = \{q_f\}$ and $\Delta = \{a \to q_a, g(q_a, q_a) \to q_f\}$. Hence $\mathcal{L}(\mathcal{A}) = \{g(a,a)\}$ and $\mathcal{R}^*(E) = \{g(f^n(a), f^n(a)) \mid n \geq 0\}$. Let $N = [g(f(x), f(y)) \to z] \to [f(x) \to q_1, f(y) \to q_1]$. During the first completion step, we find a critical pair $g(q_a, q_a) \to_{\mathcal{R}} g(f(q_a), f(q_a))$ and $q(q_a, q_a) \to^*_{\mathcal{A}} q_f$. We thus have to add the transition $g(f(q_a), f(q_a)) \to q_f$ 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 $[f(x) \to q_1, f(y) \to q_1]$ gives $[f(q_a) \to q_1, f(q_a) \to q_1]$. This last system is used to normalize the transition $g(f(q_a), f(q_a)) \to q_f$ into the set $\{g(q_1, q_1) \to q_f, f(q_a) \to q_1\}$ which is added to $\mathcal{A}$ to obtain $\mathcal{A}^1_{N,\mathcal{R}}$. The completion process continues for another step and ends on $\mathcal{A}^2_{N,\mathcal{R}}$ whose set of transition* *is* $\{a \to q_a, g(q_a, q_a) \to q_f, g(q_1, q_1) \to q_f, f(q_a) \to q_1, f(q_1) \to q_1\}$. *We have* $\mathcal{L}(\mathcal{A}^k_{N,\mathcal{R}}) = \{g(f^n(a), f^m(a)) \mid n, m \geq 0\}$ *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. Timbuk also provides means to query the approximation automaton so as to achieve some reachability checks. For instance, on the previous example, once the fixpoint automaton $\mathcal{A}^k_{N,\mathcal{R}}$ has been computed, it is possible to check whether some terms are recognized or not. This can be done using tree automata intersections [7]. However, a more convenient 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 $t\sigma \to^*_{\mathcal{A}^k_{N,\mathcal{R}}} q$. 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 $s[t\sigma']_p \in \mathcal{L}(\mathcal{A}^k_{N,\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e. that $t\sigma'$ occurs as a subterm in the language recognized by $\mathcal{L}(\mathcal{A}^k_{N,\mathcal{R}})$. 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.

Example 4 *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. Using Timbuk on Example 3, we can automatically construct* $\mathcal{A}^2_{N,\mathcal{R}}$ *which can be queried using the following patterns. The answer to the pattern* $g(f(?x), f(f(f(?y))))$ *is the following set of solutions:*

```
Occurence in state qf!
solution 1: x <- qa, y <- q1          solution 2: x <- q1, y <- q1
solution 3: x <- qa, y <- qa          solution 4: x <- q1, y <- qa
```

This result means that some ground instances of this term exist in the approximation. On the opposite, the pattern $g(f(\_), g(\_, \_))$ *has no solution, meaning that no term containing any ground instance of this pattern is reachable, i.e.* $\forall u, v, w \in \mathcal{T}(\mathcal{F}) : g(a, a) \not\to_{\mathcal{R}}^* g(f(u), g(v, w))$

4 Formalisation of the Java Bytecode Semantics using Rewriting Rules

This section describes how to model an object oriented bytecode language semantics using rewriting rules. First, we show how program states are generally described in formal semantics of Java. Second, we give an encoding of this semantics using terms. Third, we present formal semantics of Java bytecode instructions. Forth, we encode the semantics of those instructions in TRS. The first and third part are based on Java semantics formalizations of the literature [3, 18, 9, 1].

4.1 Formalization of Java Program States

A Java program state contains a current frame, 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 informations that are 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 higher 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 couples (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 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 fields name 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

Now that we formally defined a program state, we need to define the set of symbols ($\mathcal{F}$, see section 2) needed to express a program state as a term. In the following, the notation $foo : i$ stands for foo is a symbol and the arity of foo is i.

We first need some set of symbols for $C_r(p) = C(p) \cup \{array\}$. This is straightforward, using names of the classes $c \in C(p)$ and $\{array\}$ as symbols with an arity 0. The corresponding sets of symbols will be referred as $\mathcal{F}_C(p)$ and $\mathcal{F}_{C_r}(p)$.

In our TRS, we encode a reference as $loc(c, a)$ where $c \in C_r(P)$ is the class of the object being referenced and a is an integer. In Java, it is always possible to know dynamically the class of an object corresponding to a reference (including a special case for arrays). This justifies that the class name c appears in the encoding of the reference itself.

- To express a primitive type (integer), we use $\mathcal{F}_{primitive} = \{succ : 1, pred : 1, zero : 0\}$.
- To express a reference, we use $\mathcal{F}_{reference}(p) = \{loc : 2, succ : 1, zero : 0\} \cup \mathcal{F}_{C_r}(p)$.

- To express a value, we use $\mathcal{F}_{value}(p) = \mathcal{F}_{primitive} \cup \mathcal{F}_{reference}(p)$.

For example, $succ(succ(zero))$ stands for the integer 2, and $loc(foo, succ(zero))$ is a reference pointing to the object located at the index 1 in the class heap dedicated to the class foo.

Now we need symbols for $PC(p)$, $M(p)$ and $M_{id}(p)$. 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 straightforward 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 $l(p)$ denotes the maximum of local variables used by the methods of the program package p. Then:

- To express an operand stack, we use $\mathcal{F}_{stack}(p) = \{stack : 2, nilstack : 0\} \cup \mathcal{F}_{value}(p)$.
- To express local variables, we use $\mathcal{F}_{localVars}(p) = \{locals : l(p), nillocal : 0\} \cup \mathcal{F}_{value}(p)$.
- To express a frame, we use $\mathcal{F}_{frame}(p) = \{frame : 4\} \cup \mathcal{F}_{PC}(p) \cup \mathcal{F}_{M_id}(p) \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{localVars}(p)$.

For example, $stack(succ(succ(zero)), stack(loc(foo, succ(zero)), nilstack))$ stands for the stack having two elements, the integer 2 at the top and a reference to an object of class foo at the bottom. $locals(loc(foo, succ(zero)), succ(succ(zero)), nilstack)$ stands for local variables where the first contains a reference and the second the integer 2. The last one isn't initialized yet ($nillocal$).

A possible frame thus would be:
$frame(name(bar, A), pp4, stack(succ(zero), nilstack), locals(loc(bar, zero), nillocal))$
The program counter points to the 4th instruction of the method bar defined by the class A. The current operand stack just have the integer 1 on the top. The first local variable is some reference and the other is not initialized.

Now we need symbols to represent the objects. 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. Let nc be the number of classes.

We chose to divide the heap into *nc class heaps* plus one for the arrays. A class heap is a list of objects of the same class. 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.

- To express a heap, we use $\mathcal{F}_{heap}(p) = \{heaps : (nc + 1), heap : 2\} \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{objects}(p)$.
- To express a state, we use $\mathcal{F}_{state}(p) = \{state : 4\} \cup \mathcal{F}_{frame}(p) \cup \mathcal{F}_{heap}(p)$.

A possible heap would be:
$heaps($
$\quad heap(succ(zero), stack(objectA, nilstack)),$
$\quad heap(succ(zero), stack(objectB(zero), nilstack)),$
$\quad nilstack)$
$)$

$$(pop)\quad \frac{(m, pc, x :: s, l)}{(m, pc+1, s, l)}$$

$$(add)\quad \frac{(m, pc, b :: a :: s, l)}{(m, pc+1, (a+b) :: s, l)}$$

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

Figure 1: example of bytecodes operating at the frame level

$$(getField_{name})$$
$$\frac{((m, pc, ref :: s, l), fs, h, k), f = getf(name, ref, h, k)}{(m, pc+1, f :: s, l)}$$

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

Figure 2: example of bytecodes operating at the state level

4.3 Java Bytecode Semantics

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

Figure 1 presents some rules of the semantics operating at the frame level. 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 semantics of the first two instructions is straightforward, the $store_i$ instruction is used to store the value at the top of the current stack in the i^{th} register, where $x \rightarrow_i l$ denotes the new resulting array of local variables.

Figure 2 presents some rules 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 awaiting frames, h the current heap and k the current static heap.

The $getField_{name}$ instruction push on the stack the value stored in the field $name$ of the object whose reference is at the top of the stack. The internal function $getf(name, ref, h, k)$ is used to access this value in the corresponding heap.

$invokeVirtual_{name}$ is used to invoke a method in a dynamic way. This method is chosen 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 $lookup(name, c)$ to get the method identifier m' corresponding to this name and this class. We also need two more 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

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	$next(pp2)$	$\rightarrow$	$pp3$

Figure 3: *pop* instruction by rewriting rules

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.

4.4 Java Bytecode Semantics using Rewriting Rules

In this section we analyse 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). Another problem is that one step in an operational semantics involves one visible step and many intermediate ones modeling internal functions (such as $getf$, $getParams$, ...). However, these intermediate steps 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 now show how to express the examples of the Java Bytecode Semantics in section 4.3 by means of rewriting rules.

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. Then it is possible to write generic rewriting rules for the instructions of the semantics. If an instruction requires several internal rewriting steps, we will only rewrite the corresponding $xframe$ term until the execution of the instruction ends.

We take the example of a *pop* instruction at the second program point in a method A of a class foo (Figure 3). Rule 1 builds a $xframe$ term by explicitly adding the current instruction to the $frame$ term. Rule 2 describes pop's semantics. Rule 3 is trivial and allows to get to next program point.

Our goal is to build the set of reachable program state as precisely as we can. With regards to this aspect, using two different symbols presents two advantages. First, a $frame$ term always represents a "real" program state, thus making easier to browse the result of our analysis. Second, since we ensure that all internal steps are enclosed in $xframe$ terms, $frame$ terms do not contain partially evaluated terms, thus avoiding non-determinism of rewriting. To illustrate this, Figure 4 presents a wrong way to represent *add* instruction's semantics. The main problem in this one comes from rule 2, where the computation of *xadd* is carried out in the next $frame$ term. It produces a lot of $frame$ terms for a single program point and thus creates a lot of possible rewriting branches.

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow$	$xframe(add, name(foo, A), pp2, s, l)$
2	$xframe(add, m, pc, stack(b, stack(a, s)), l)$	$\rightarrow$	$frame(m, next(pc), stack(xadd(a, b), s), l)$
3	$xadd(succ(a), succ(b))$	$\rightarrow$	$xadd(a, succ(succ(b)))$
4	$xadd(succ(a), pred(b))$	$\rightarrow$	$xadd(a, b)$
5	$xadd(pred(a), pred(b))$	$\rightarrow$	$xadd(a, pred(pred(b)))$
6	$xadd(pred(a), succ(b))$	$\rightarrow$	$xadd(a, b)$
7	$xadd(zero, b)$	$\rightarrow$	b
8	$xadd(a, zero)$	$\rightarrow$	a
9	$next(pp2)$	$\rightarrow$	$pp3$

Figure 4: *add* instruction by rewriting rules, first way

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow$	$xframe(add, name(foo, A), pp2, s, l)$
2	$xframe(add, m, pc, stack($ $succ(b), stack(a, s)), l)$	$\rightarrow$	$xframe(xadd(a, b), m, pc, s, l)$
3	$xframe(result(x), m, pc, s, l)$	$\rightarrow$	$frame(m, next(pc), stack(x, s), l)$
4	$xadd(succ(a), succ(b))$	$\rightarrow$	$xadd(a, succ(succ(b)))$
5	$xadd(succ(a), pred(b))$	$\rightarrow$	$xadd(a, b)$
6	$xadd(pred(a), pred(b))$	$\rightarrow$	$xadd(a, pred(pred(b)))$
7	$xadd(pred(a), succ(b))$	$\rightarrow$	$xadd(a, b)$
8	$xadd(zero, b)$	$\rightarrow$	$result(b)$
9	$xadd(a, zero)$	$\rightarrow$	$result(a)$
10	$next(pp2)$	$\rightarrow$	$pp3$

Figure 5: *add* instruction by rewriting rules, second way

Figure 5 presents a more deterministic way to do it. Once the $frame$ term is rewritten to a $xframe$ term (rule 1), the next step (rule 2) use a rewriting sub-system $xadd$ (rules 4 to 9). It works the same as previous example, but this time we use a new constructor ($result$) to store the result of the addition. Then rule 3 puts it on the top of the stack. This way we switch from $xframe$ to $frame$ context when the internal computation of the 'add' instruction ends, thus making sure we do not produce useless $frame$ terms.

So far, all the rewriting rules are generic, i.e. do not depend on the bytecode program. Let us now give some rules depending on the program p itself. The $store_i$ instruction is a good example. Since the way we handle local variables depends on $l(p)$ (maximal number of local variables in p, see section 4.2) it will vary from a program to another. In our example, we assume that $l(p) = 3$, see Figure 6. Note that we introduce as many ground terms ($local0 : 0$, $local1 : 0$ and $local2 : 0$) and variables ($l0$, $l1$ and $l2$) as $l(p)$.

1	$frame(name(foo, A), pp2, s, l)$	$\to$	$xframe(store(local0), pp2, name(foo, A), s, l)$
2	$xframe(store(local0), pc, m,$ $stack(x, s), locals(l0, l1, l2))$	$\to$	$frame(next(pc), m, s, locals(x, l1, l2))$
3	$xframe(store(local1), pc, m,$ $stack(x, s), locals(l0, l1, l2))$	$\to$	$frame(next(pc), m, s, locals(l0, x, l2))$
4	$xframe(store(local2), pc, m,$ $stack(x, s), locals(l0, l1, l2))$	$\to$	$frame(next(pc), m, s, locals(l0, l1, x))$
5	$next(pp2)$	$\to$	$pp3$

Figure 6: $store_i$ instruction by rewriting rules

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

Figure 7: $invokeVirtual_{name}$ instruction by rewriting rules

The $invokevirtual_{name}$ instruction raises another problem. This time we need to compile some information about methods and 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. Let us consider two classes, A and B, B extending 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 7 presents the resulting rules for this simple example. The choice we made about heap representation is beneficial here, since the class of the object on the top of the stack is directly available in the reference itself. Here we can take advantage of this and avoid to encode the lookup function by rewriting.

The last instruction we present here, $getField$, loads the value of a field of an object stored in the heap. We use a sub rewriting system to extract the object, we are looking for, from the corresponding heap. We consider a small example of a class A with no fields and a class B with a field $field0$. In our setting, the heap is made of 3 different heaps here. The

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow$	$xframe(getField(f), name(foo, 1), pp2, s, l)$
2	$state(xframe(getField(field0),$ $m, pc, stack(loc(B, adr), s), l),$ $fs, heaps(h0, heap(size, h1), ha), k)$	$\rightarrow$	$state(xframe(xGetField(B, field0, h1,$ $size, adr), m, , pc, s, l), fs,$ $heaps(h0, heap(size, h1), ha), k)$
3	$xframe(xGetField(B, field0$ $, stackB(objectB(x), h1),$ $succ(zero), zero), m, pc, s, l)$	$\rightarrow$	$frame(m, next(pc), stack(x, s), l)$
4	$xGetField(B, field0, h1,$ $succ(size), succ(adr))$	$\rightarrow$	$xGetField(B, field0, h1, size, adr)$
5	$xGetField(B, field0, stackB(x, h1),$ $succ(succ(size)), zero)$	$\rightarrow$	$xGetField(B, field0, h1, succ(size), zero)$
6	$next(pp2)$	$\rightarrow$	$pp3$

Figure 8: $getField$ instruction by rewriting rules

first one is for class A, the second for class B and the last one for arrays (See Figure 8). Rule 2 extracts from the general heap the one corresponding to class A. Rules 4 and 5 locate the object corresponding to the address. Rules 3 put the value of the field we were looking for, on the stack.

Note that, modeling the semantics and the program by rewriting rules is not enough to build an analysis. We also need stubs for native libraries used by the program. As an example, we implement one for some of the methods from the $javaioInputStream$ 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 using Rewriting Theory

In most program analysis, it is often necessary to know the control flow graph. For Java, as for all 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 9, `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 very example, exceptions around `System.in.read()` are required by the Java compiler. However, at present, we do not take them into account in the control flow.

There are different standard class analysis, from simple and fast to precise and expensive. We consider k-CFA analysis [17]. For those analysis, primitive types are abstracted by the name of their type and references are abstracted by the class of the objects they points to.

```
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;                            }
```

Figure 9: Java Program Example

In 0-CFA analysis, each method is analyzed only once and, thus, merges the parameters of the different calls to this method. k-CFA is able to distinguish between different calls to the same method by 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 Java program, a tree automaton $\mathcal{A}$ recognizing a set of initial Java program states, we aim at computing an automaton $\mathcal{A}^k_{N,\mathcal{R}}$ over approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. We developed a prototype which produces $\mathcal{R}$ and $\mathcal{A}$ from a Java `.class` file. For the file `Test.class`, generated by the compilation of the Java program of Figure 9, $\mathcal{R}$ is composed of 275 rewrite rules. The analysis itself is performed using Timbuk [13]. Successively, this section details a 0-CFA, a 1-CFA and an even more precise analysis achieved 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 goes on. Moreover, since we want to analyse 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 $\mathcal{A}$: $zero \to q_{int}$, $succ(q_{int}) \to q_{int}$ and $pred(q_{int}) \to q_{int}$. The input stream is also specified by $\mathcal{A}$ as being an infinite stack of integer. This can be done thanks to the following transitions: $nilstackin \to q_{in}$ and $stackin(q_{int}, q_{in}) \to q_{in}$. Approximation rules for integers, stream and references are defined in the following way: $[x \to y] \to [zero \to q_{int}, succ(q_{int}) \to q_{int}, pred(q_{int}) \to q_{int}, nilstackin \to q_{in}, stackin(q_{int}, q_{in}) \to q_{in}, loc(A, \alpha) \to q_{refA}, loc(B, \beta) \to q_{refB}]$ where x, y, α and β are variables. The pattern

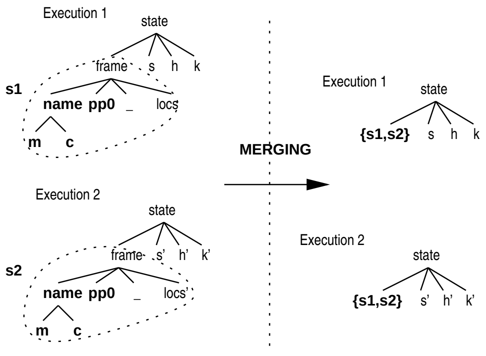

Figure 10: Principle of approximation rules for a 0-CFA analysis

$[x \to y]$ matches any new transition to normalize and the rules $loc(A, x) \to q_{refA}$ and $loc(B, y) \to q_{refB}$ 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.

The approximation rules for frames and states are built according to the principle illustrated in Figure 10. Independently of the current state of the execution in which the method `m` of the class `c` is invoked, the frames representing two different calls to `m` are merged. 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 $\mathcal{A}^{145}_{N,\mathcal{R}}$ 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 exists two possible values for $?c$: q_A and q_B which are the states recognizing respectively the classes A and B. This is coherent with 0-CFA which is not able to discriminate between the two possible calls to the *execute* method. Note that, for analysing unbounded recursive calls, the regular nature of approximations allows to finitely abstract the infinite frame stacks. This is however not detailed in this paper.

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, input stream and references are similar to the ones used for 0-CFA. In N',

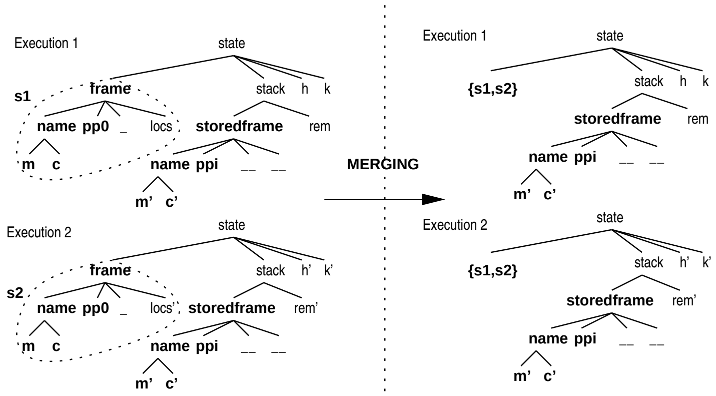

Figure 11: Principle of approximation rules for a 1-CFA analysis

approximation rules for states and frames are designed according to the principle illustrated in Figure 11. Contrary to Figure 10, 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 a complete set of approximation rules N'. Using N', completion terminates on $\mathcal{A}^{140}_{N',\mathcal{R}}$. 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 a more precise result than 0-CFA, since each pattern as only one solution for $?c$: q_A for the first and q_B for the second. Note that using a similar pattern to query $\mathcal{A}^{145}_{N,\mathcal{R}}$, the 0-CFA automaton, gives q_A and q_B as solution for $?c$ for both program points.

5.3 Getting more precise when necessary

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 analysis, integers are abstracted by their type. One of the advantage of our technique is its ability to easily make approximation more precise by retrieving some approximation rules.

We can now refine the approximation of the naturals by distinguishing between 0, 1, 2 and 3 or more. This can be done using the following transitions: $0 \to q_0$, $succ(q_0) \to q_1$,

$succ(q_1) \rightarrow q_2$, $succ(q_2) \rightarrow q_{int}$ and $succ(q_{int}) \rightarrow q_{int}$. For specifying the negative integers, the following transitions are used: $pred(q_0) \rightarrow q_{negint}$ and $pred(q_{negint}) \rightarrow q_{negint}$. The input stream representation is also modified by the following transitions: $nilstackin \rightarrow q_{in}$, $stackin(q_{negint}, q_{in}) \rightarrow q_{in}$, $stackin(q_{int}, q_{in}) \rightarrow q_{in}$ and $stackin(q_j, q_{in}) \rightarrow q_{in}$ with $j = 0, \ldots, 2$.

In fact, no other approximation is needed to ensure termination of the completion. On the fixpoint automaton, 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 the last control point of the program.

This result is not surprising. The first result is possible when there is zero iteration 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 was impossible to obtain with the two previous analysis presented in Section 5.1 and 5.2.

6 Related works

Term rewriting systems have been used to define and prototype semantics for a long time. However, this subject has recently raised up for a verification purpose. In [6, 16], rewriting is also used as operational semantics for Java. However, 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. On the opposite, 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 have to commute. Such property seems hard to prove on a TRS encoding the Java semantics.

Takai [19] also proposed a theoretical version of approximated reachability analysis over term rewriting systems, without the left-linearity restriction we have. 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.

Compared to classical static classes analysis (like in [17]), depending of the approximation rules we need to ensure termination of our algorithm, we can get the several ranges of precision of k-CFA. Starting from an automatically generated approximation, it is also possible to adapt approximation rules so as to get a more precise abstraction and prove specific properties on a program.

7 Conclusion

We have defined a technique for producing 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 set of reachable states is not exactly computable in general so we use approximations to force the computation of $\mathcal{R}^*(E)$ to terminate. The advantage of this approximation technique is that its safety is guaranteed by the underlying theory and does not have to be proved for each proposed abstraction.

Approximation rules defining usual analysis, such as k-CFA analysis, can clearly be automatically produced from the program source. We are currently experimenting some other analysis where the regular nature of the approximation could be beneficial. First to come to mind are shape analysis and race detection in multi-threaded programs. In addition, we expect to have the same precision as symbolic evaluation techniques, like in [8], with cautious approximation generation.

References

[1] I. Attali and a Denis. A formal executable semantics for java, 1990.

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

[3] P. Bertelsen. Semantics of Java Byte Code. Technical report, 1997.

[4] 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.

[5] 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.

[6] 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.

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

[8] Thomas Jensen Frédéric Besson and David Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 2006.

[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] I. Siveroni. Operational semantics of the java card virtual machine, 2004.

[19] 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.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399