Cibai: An Abstract Interpretation-Based Static Analyzer for Modular Analysis and Verification of Java Classes

Francesco Logozzo

École Normale Supérieure
45, rue d'Ulm, Paris, France
Francesco.Logozzo@Polytechnique.edu

Abstract. We introduce Cibai a generic static analyzer based on abstract interpretation for the modular analysis and verification of Java classes. We present the abstract semantics and the underlying abstract domain, a combination of an aliasing analysis and octagons. We discuss some implementation issues, and we compare Cibai with similar tools, showing how Cibai achieves a higher level of automation and precision while having comparable performances.

1 Introduction

Object-oriented programming emphasizes the development by components. Components are written once and used in many, different contexts. Component reliability is a main issue in object-oriented development.

Testing has been for long time the main approach for assuring component's reliability. A popular approach is that of unit testing, *e.g.* JUnit [1], which allows to write test cases for single components, and then to "*validate*" the tests through the use of assertions. The problems of the approach are that: (i) it requires the programmers to write test cases (ii) it is not sound as just finitely many execution paths and inputs can be considered; and (iii) it does not scale up very well as, if one wants full code coverage, the complexity of the test cases grows up very quickly with the size of the program. As a consequence, the need for formal methods arises.

Most of the verification tools that have been developed so far heavily relies on program annotations, *e.g.* [5,17,3]. Following the Design by Contract approach [18], such tools allow the programmer to express class invariants, pre-condition, post-conditions for the class. From the annotated program they derive the verification conditions, which are passed to a theorem prover. This approach has two main problems: (i) it has an inherent exponential behavior, as the checking of verification conditions by the theorem prover roughly corresponds to the exploration of all the possible paths in the program; and (ii) it requires the developer to provide inductive arguments, *e.g.* loop invariants, either as further annotations to the source code, or during the interactive proof.

B. Cook and A. Podelski (Eds.): VMCAI 2007, LNCS 4349, pp. 283–298, 2007.

We believe that in order to have a practical interest, a tool for the verification of object-oriented programs must be automatic, or it must require the least possible amount of interaction with the user. We have developed a tool, Cibai, based on abstract interpretation [6], for the analysis of Java classes. The tool analyzes "*pure*" Java classes, *i.e.* it does not require any annotation. It infers class invariants, loop invariants and method postconditions. The inferred properties are then used for verifying the absence of run-time errors in the class. Currently we can verify (i) the absence of divisions by zero; (ii) the absence of accesses out of the bounds of arrays; (iii) the absence of null dereferences; and (iv) simple user-provided assertions.

As an example consider the class `MiniBag` in Fig. 1. Cibai can discover in few milliseconds the class invariant $0 \leq$ `top` $\leq$ `elements.length`. It discovers that the array creation at line $(*)$ may launch an exception when `initial` is negative. It uses the inferred class invariant to prove that all the array accesses in the body of the methods are correct, *i.e.* no exception is thrown. Unlike existing tools, as ESC/Java or Spec#, it does require no annotation nor interaction with the user.

Paper organization. Section 2 recalls some notation and results from [15]. Section 3 describes the abstract domains designed for and implemented in Cibai. Section 4 presents the structure of the analyzer and the description of the most interesting transfer functions. Section 5 reports some experience with the tool. Section 6 compares Cibai with related tools and Section 7 conclude the paper.

2 Preliminaries

Syntax. In order to simplify the presentation, we perform some simplifying assumptions on the syntax of programs. We assume that a class has a unique constructor and that all the fields are protected. We also omit access modifiers in the definition of fields and methods. Nevertheless, in our implementation we handle those cases. We assume that all the fields are typed (as it is the case in mainstream object-oriented languages as Java or C#) and we also assume the basic types to be just `int` and `boolean`. We omit here the details of the statements that constitute the body of the class constructor and of the methods, too.

A class `C` is a tuple $\langle$`init`, `F`, `M`$\rangle$ where `init` is the class constructor, `F` is a set of field declarations and `M` is a set of methods.

Concrete Semantic Domains. We model an execution state of a Java program with a pair made up of an environment and a state An environment is a map from variables to memory addresses. A store is a map from addresses to values. Values are either basic values (`int`s, `boolean`s), the `void` value ϕ or references. A reference is a pair made up of a type and an environment.

The internal environment of an object is stored at a given memory location. The address corresponding to such a location is the *identity* of the object. In the following, we will denote the set of all the concrete states by Σ.

```
class BoundError extends Throwable { }
class MiniBag {
    private int[] elements;
    private int top;

    MiniBag(int initial) {
(*)  top = 0; elements = new int[initial]; }

    int remove() throws BoundError {
     int r;
     if(top > 0 ) {
       top--; r = elements[top];
     } else throw new BoundError();
     return r; }

    void add(int i) {
      if(top < elements.length) {
        elements[top] = i; top++;
      } else throw new BoundError(); }

    void removeMinimum() {
      if(top > 0) {
        int min = 0, i;
        for(i = 1; i < top; i++)
          if(elements[min] > elements[i])
            min = i;
        elements[min] = elements[i-1]; top--; } } }
```

Fig. 1. A bag of `int`. Cibai emits a warning at $(*)$ as `initial` may be negative. It proves correct all the other array accesses. Unlike ESC/Java 2 or Spec# it does not require the user to provide annotations for the class invariant and for the loop invariant of `removeMinimum`.

Abstract Semantics. Let $\bar{\mathsf{D}}$ be an abstract domain $\bar{\mathsf{D}}$ related to $\mathcal{P}(\Sigma)$ by a monotonic concretization function γ such that $\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \xleftarrow{\gamma} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$. As a consequence, we drop the need for the best possible abstraction of concrete elements, putting ourselves in a relaxed abstract interpretation framework, [7].

We recall from [15, §5] the fixpoint formulation of class invariants:

Proposition 1 (Abstract class invariant). *Let* $\mathtt{C}$ *be a class. Let* $\mathbb{I}[\![\mathtt{init}]\!] \in \mathcal{P}(\Sigma)$ *be the collecting semantics of the constructor, let* $\mathbb{M}[\![\cdot]\!] \in [\mathtt{M} \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]$ *be the collecting semantics of the methods, and let* $\mathrm{Context}(\mathtt{C}) \in \mathcal{P}(\Sigma)$ *be the collecting semantics for the behavior of the context, [15].*

Let $\bar{\mathbb{I}}[\![\mathtt{init}]\!] \in \bar{\mathsf{D}}$ *be a sound approximation for the constructor's semantics:* $\mathbb{I}[\![\mathtt{init}]\!] \subseteq \gamma(\bar{\mathbb{I}}[\![\mathtt{init}]\!])$. *For each* $\mathtt{m} \in \mathtt{M}$ *let* $\bar{\mathbb{M}}[\![\mathtt{m}]\!] \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ *be a sound approximation of its semantics:*

$$\forall S \in \mathcal{P}(\Sigma).\ \forall \bar{S} \in \bar{\mathsf{D}}.\ S \subseteq \gamma(\bar{S}) \Longrightarrow \mathbb{M}[\![\mathtt{m}]\!](S) \subseteq \gamma(\bar{\mathbb{M}}[\![\mathtt{m}]\!](\bar{S})).$$

Finally, let $\overline{\text{Context}} \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ *be a sound approximation of the context behavior:* $\forall S \in \mathcal{P}(\Sigma).\ \forall \bar{S} \in \bar{\mathsf{D}}.\ S \subseteq \gamma(\bar{S}) \Longrightarrow \text{Context}(S) \subseteq \gamma(\overline{\text{Context}}(\bar{S}))$. *Then*

$$\bar{\mathbb{C}}[\![\mathtt{C}]\!] = \text{lfp}^{\bar{\sqsubseteq}}_{\bar{\bot}} \lambda X.\ \bar{\mathbb{I}}[\![\mathtt{init}]\!] \bar{\sqcup} \bar{\bigsqcup}_{\mathtt{m} \in \mathtt{M}} \bar{\mathbb{M}}[\![\mathtt{m}]\!](X) \bar{\sqcup} \text{Context}(X) \tag{1}$$

is such that $\mathbb{C}[\![\mathtt{C}]\!] \subseteq \gamma(\bar{\mathbb{C}}[\![\mathtt{C}]\!])$.

Please note that, as we do not restrict ourselves to abstract domains that respect the ascending chain condition (ACC) we need of a widening operator to upper approximate the fixpoint in (1).

3 Abstract Domains

In Cibai we chose $\bar{\mathsf{D}}$ to be the abstract domain $\overline{\mathsf{Env}} \times \overline{\mathsf{Store}} \times \mathcal{P}(\overline{\mathsf{Addr}})$. $\overline{\mathsf{Env}}$ is a map between variables and sets of abstract addresses. $\overline{\mathsf{Store}}$ is the (reduced) Cartesian product of the abstraction for integers, basic values and references. Finally, $\mathcal{P}(\overline{\mathsf{Addr}})$ is the set of the abstract addresses, corresponding to class fields, which may escape from the class context.

3.1 Abstract Environment

Intuitively, an abstract address stands for one of the two: a single address or a possibly infinite set of addresses. We require that it exists a partition of $\overline{\mathsf{Addr}}$ into two disjoint sets such that in the concrete the elements of one does not overlap with those of the other set. More formally:

Definition 1 (Abstract addresses, $\overline{\mathsf{Addr}}$). *Let* $\mathcal{P}(\mathsf{Addr})$ *be a set of addresses. Let* $\overline{\mathsf{Addr}}$ *be a set, and* $\gamma_{\mathsf{a}} \in [\mathcal{P}(\overline{\mathsf{Addr}}) \to \mathcal{P}(\mathsf{Addr})]$ *a monotonic concretization function. Then, we say that* $\mathcal{P}(\overline{\mathsf{Addr}})$ *is a set of abstract addresses if it can be partitioned into two disjoint subsets* $\mathcal{P}(\overline{\mathsf{Addr}}_e)$ *(exact addresses) and* $\mathcal{P}(\overline{\mathsf{Addr}}_s)$ *(summary addresses) such that:*

- $\forall \bar{\mathsf{a}} \in \mathcal{P}(\overline{\mathsf{Addr}}_e).\ \exists \mathsf{a} \in \mathcal{P}(\mathsf{Addr}).\ \gamma_{\mathsf{a}}(\bar{\mathsf{a}}) = \{\mathsf{a}\}$
- $\forall \bar{\mathsf{a}} \in \mathcal{P}(\overline{\mathsf{Addr}}_s).\ \gamma_{\mathsf{a}}(\bar{\mathsf{a}}) = A \Longrightarrow \forall A'.A \neq A' \wedge (\exists \bar{\mathsf{a}'} \in \mathcal{P}(\overline{\mathsf{Addr}}_s).\ \gamma_{\mathsf{a}}(\bar{\mathsf{a}'}) = A') \Longrightarrow A \cap A' = \emptyset$.

Please note that we do not require to have the *best* approximation for a set of concrete addresses, so that once again we use a relaxed abstraction interpretation framework.

An abstract environment tracks, for each variable the set of addresses it may point to. We define it as a map from variables to sets of abstract addresses. The abstract operations on such an abstract domain are defined as the point-wise and functional extension of those on $\mathcal{P}(\overline{\mathsf{Addr}})$.

Definition 2 (Abstract environment, $\overline{\mathsf{Env}}$). *The domain of abstract environments is* $\langle[\mathtt{Vars} \to \mathcal{P}(\overline{\mathsf{Addr}})], \bar{\sqsubseteq}_\mathsf{e}, \bar{\bot}_\mathsf{e}, \bar{\top}_\mathsf{e}, \bar{\sqcup}_\mathsf{e}, \bar{\sqcap}_\mathsf{e}\rangle$. *The relation with the concrete is given by a monotonic map* $\gamma_\mathsf{e} \in [\overline{\mathsf{Env}} \to \mathcal{P}(\mathsf{Env})]$ *defined as*

$$\gamma_\mathsf{e} = \lambda\bar{\mathsf{e}}.\ \{\mathsf{e} \in \mathcal{P}(\mathsf{Env}) \mid \forall \mathtt{x}.\ \mathsf{e}(\mathtt{x}) \in \gamma_\mathsf{a}(\bar{\mathsf{e}}(\mathtt{x}))\}.$$

3.2 Abstract Store

The abstract store must approximate (i) basic values as booleans or integers; and (ii) reference values.

Abstraction of basic values. We approximate all the basic values but integers using a non-relational abstraction.

We chose booleans as representative for the abstraction of non-integer basic values. Booleans are approximated by an abstract domain $\bar{\mathsf{B}} = [\mathsf{Addr} \to \{\bar{\top}, \mathtt{true}, \mathtt{false}, \bar{\bot}\}]$. The abstract operations (join, meet, widening, etc.) are defined point-wise. The concretization function, $\gamma_\mathsf{b} \in [\bar{\mathsf{B}} \to \mathcal{P}(\mathsf{Store})]$ is straightforward.

For integers, we have implemented two different abstract domains. The first one is that of Intervals [6], in which we abstract each location corresponding to an integer value (`int`-location) with the lower and the upper bounds for the values that can be stored in such location. This one gives excellent performances, but in practice we found it to be too imprecise for the purposes of our analysis. The second one is the abstract domain of octagons [19]. We recall that the Octagon abstract domain captures relations in the form of $\pm\mathtt{x} \pm \mathtt{y} \leq \mathtt{k}$, where `x` and `y` are identifiers and `k` is a numeric constant. As a consequence, the Octagon abstract domain allow us to keep relations between different `int`-locations, so to achieve a greater precision yet keeping good performances. We do not recall here the order, the join, the meet and the widening on octagons. However, as we use it later, we recall the concretization function:

Definition 3 (Octagon concretization, γ_o). *Let* Id *be a set of identifiers. We denote by* $\mathsf{Octagon}_\mathsf{Id}$ *the set of octagons constraints built on the top of* Id. *Then* $\gamma_\mathsf{o} \in [\mathsf{Octagon}_\mathsf{Id} \to (\mathsf{Id} \to \mathbb{N})]$ *is defined as*

$$\gamma_\mathsf{o} = \lambda\mathsf{oct}.\ \{\sigma \mid \forall \mathsf{id}_1, \mathsf{id}_2 \in \mathsf{Id}. \forall s_1, s_2 \in \{0, +, -\}.\\ s_1 * \mathsf{id}_1 + s_2 * \mathsf{id}_2 \leq \mathtt{k} \in \mathsf{oct} \implies s_1 \cdot \sigma(\mathsf{id}_1) + s_2 \cdot \sigma(\mathsf{id}_2) \leq k\}.$$

In the following, we will often write $\mathsf{Octagon}_n$ to denote an octagon with n distinct identifiers or simply Octagon when neither the identifiers nor the dimensions of the octagon are relevant to the context.

In the context of our analyzer we have to pay attention to (i) the fact that not all the addresses correspond to a dimension in the octagon; and (ii) the size of the octagons may dynamically change, because of dynamic memory allocation. As a consequence, we map each address corresponding to an `int`-location to a dimension in the octagon. We lift the usual operations on octagons so to handle,

e.g. the join of two octagons of different sizes. We postpone the detailed description of such operations to the Section 4.2, as they involve several implementation details. The concretization of a dynamic octagon is the set of all the concrete stores where the octagon constraints are satisfied by the `int`-locations.

Definition 4 (Dynamic octagons, $\mathsf{DynOctagon}$). *Let* Id *be a set of identifiers. The abstract domain of dynamic octagons is*

$$\mathsf{DynOctagon}_{\mathsf{Id}} = \langle [\overline{\mathsf{Addr}} \to \mathsf{Id}] \times \mathsf{Octagon}_{\mathsf{Id}}, \bar{\sqsubseteq}_{\mathsf{do}}, \bar{\bot}_{\mathsf{do}}, \bar{\top}_{\mathsf{do}}, \bar{\sqcup}_{\mathsf{do}}, \bar{\sqcap}_{\mathsf{do}} \rangle.$$

The meaning of a dynamic octagon is given by the monotonic function $\gamma_{\mathsf{do}} \in [\mathsf{DynOctagon}_{\mathsf{Id}} \to \mathcal{P}(\mathsf{Store})]$ *defined as*

$$\begin{aligned}\tilde{\gamma} &= \lambda\langle f, \mathsf{oct}\rangle.\ \{\sigma \in [\overline{\mathsf{Addr}} \to \mathbb{N}] \mid \sigma' \in \gamma_{\mathsf{o}}(\mathsf{oct}),\ \sigma = \sigma' \circ f\},\\ \gamma_{\mathsf{do}} &= \lambda\langle f, \mathsf{oct}\rangle.\ \{\mathsf{s} \in \mathsf{Store} \mid \forall \bar{\mathsf{a}} \in \mathrm{dom}(f).\ \sigma \in \tilde{\gamma}(\langle f, \mathsf{oct}\rangle)\\ &\qquad\qquad \Longrightarrow \exists \mathsf{a} \in \gamma_{\mathsf{a}}(\bar{\mathsf{a}}).\ \mathsf{s}(\mathsf{a}) = \sigma(\bar{\mathsf{a}})\}.\end{aligned}$$

Abstraction of reference values. We chose to approximate reference values with a pair made up of a set of reference types, *i.e.* the possible dynamic types of the reference, and an abstract environment. We "*squeeze*" together all the references that can be stored at a given address. In particular if we have o_B instance of a class `Base` and o_S instance of a class `Sub`, with `Sub` subclass of `Base`, and both o_B and o_S may be stored at the same address, then (i) the dynamic type of the abstract reference is $\{\mathtt{Base}, \mathtt{Sub}\}$; and (ii) the abstract environment is an over-approximation of the union of the two environments.

The domain of abstract references is defined below, where the domain operations are the functional and point-wise extension on operations on sets and abstract environments.

Definition 5 (Abstract references, $\overline{\mathsf{Ref}}$). *Let* `RType` *be the set of reference types. The abstract domain of abstract references is*

$$\overline{\mathsf{Ref}} = \langle [\overline{\mathsf{Addr}} \to (\mathcal{P}(\mathtt{RType}) \times \overline{\mathsf{Env}})], \bar{\sqsubseteq}_{\mathsf{r}}, \bar{\bot}_{\mathsf{r}}, \bar{\top}_{\mathsf{r}}, \bar{\sqcup}_{\mathsf{r}}, \bar{\sqcap}_{\mathsf{r}} \rangle.$$

The meaning is given by the monotonic function $\gamma_{\mathsf{r}} \in [\overline{\mathsf{Ref}} \to \mathcal{P}(\mathsf{Store})]$ *defined as*

$$\begin{aligned}\gamma_{\mathsf{r}} = \lambda f.\ \{\mathsf{s} \in \mathsf{Store} \mid \forall \bar{\mathsf{a}} \in \mathrm{dom}(f).\ f(\bar{\mathsf{a}}) = \langle \mathtt{T}, \bar{\mathsf{e}}\rangle \wedge \mathsf{a} \in \gamma_{\mathsf{a}}(\bar{\mathsf{a}}) \wedge \mathsf{s}(\mathsf{a}) = \langle \mathtt{t}, \bar{\mathsf{e}}\rangle\\ \Longrightarrow \mathtt{t} \in \mathtt{T} \wedge \mathsf{e} \in \gamma_{\mathsf{e}}(\bar{\mathsf{e}})\}.\end{aligned}$$

Abstract Store. To recapitulate, an abstract store is a non-relational abstraction of basic values and references. We use a relational abstraction for integers, and a non-relation for all the other basic values. Furthermore as Java is a strongly typed language, we can partition the (abstract) addresses depending to the type of the locations they refer to. So if $\langle \bar{\mathsf{b}}, \langle f_o, \mathsf{oct}\rangle, \bar{\mathsf{r}}\rangle$ is an element of $\bar{\mathsf{B}} \times \mathsf{DynOctagon} \times \overline{\mathsf{Ref}}$, then

$$\mathrm{dom}(\bar{\mathsf{b}}) \cap \mathrm{dom}(f_o) = \emptyset \wedge \mathrm{dom}(\bar{\mathsf{b}}) \cap \mathrm{dom}(\bar{\mathsf{r}}) = \emptyset \wedge \mathrm{dom}(f_o) \cap \mathrm{dom}(\bar{\mathsf{r}}) = \emptyset. \quad (2)$$

The definition of the operations on the domain of abstract stores are as usual the point-wise extension of the operations of the components.

Definition 6 (Abstrace store, $\overline{\mathsf{Store}}$). *The domain of abstract stores is*

$$\overline{\mathsf{Store}} = \langle \bar{\mathsf{B}} \times \mathsf{DynOctagon} \times \overline{\mathsf{Ref}}, \bar{\sqsubseteq}_{\mathsf{s}}, \bar{\bot}_{\mathsf{s}}, \bar{\top}_{\mathsf{s}}, \bar{\sqcup}_{\mathsf{s}}, \bar{\sqcap}_{\mathsf{s}} \rangle$$

whose elements satisfy (2). The concretization $\gamma_{\mathsf{s}} \in [\overline{\mathsf{Store}} \to \mathcal{P}(\mathsf{Store})]$ *is defined as* $\gamma_{\mathsf{s}} = \lambda\langle \bar{\mathsf{b}}, \bar{\mathsf{d}}, \bar{\mathsf{r}} \rangle.\ \gamma_{\mathsf{b}}(\bar{\mathsf{b}}) \cap \gamma_{\mathsf{do}}(\bar{\mathsf{d}}) \cap \gamma_{\mathsf{r}}(\bar{\mathsf{r}})$.

3.3 Abstract State

An abstract state is a non-relational abstraction of a set of interaction states. It is a triple made up of an abstract environment, and abstract store and an abstraction of the addresses which escapes from a class. We refer the interested reader to [16] for an extensive description of escaping addresses.

Proposition 2 (Abstract state, $\bar{\Sigma}$). *The domain of abstract states is*

$$\bar{\Sigma} = \langle \overline{\mathsf{Env}} \times \overline{\mathsf{Store}} \times \mathcal{P}(\overline{\mathsf{Addr}}), \bar{\sqsubseteq}_{\bar{\Sigma}}, \bar{\bot}_{\bar{\Sigma}}, \bar{\top}_{\bar{\Sigma}}, \bar{\sqcup}_{\bar{\Sigma}}, \bar{\sqcap}_{\bar{\Sigma}} \rangle.$$

The concretization function is the monotonic function $\gamma_{\bar{\Sigma}} \in [\bar{\Sigma} \to (\mathcal{P}(\Sigma) \times \mathcal{P}(\mathsf{Addr}))]$ *defined as*

$$\gamma_{\bar{\Sigma}} = \lambda\langle \bar{\mathsf{e}}, \bar{\mathsf{s}}, \bar{\mathsf{Esc}} \rangle.\ \langle \{ \langle \mathsf{e}, \mathsf{s} \rangle \mid\ \mathsf{e} \in \gamma_{\mathsf{e}}(\bar{\mathsf{e}}),\ \mathsf{s} \in \gamma_{\mathsf{s}}(\bar{\mathsf{s}}) \}, \cup_{\bar{\mathsf{a}} \in \bar{\mathsf{Esc}}}\ \gamma_{\mathsf{a}}(\bar{\mathsf{a}}) \rangle\ .$$

As a consequence, $\bar{\Sigma}$ *is a sound abstraction of* $\mathcal{P}(\Sigma)$.

4 The Analyzer

4.1 Overall Structure

Our analyzer takes as input a Java compilation unit, [10]. A compilation unit is a bunch of interface and class definitions.

First, Cibai parses the compilation unit, determines the dependencies between the different classes and interfaces. A class/interface `A` depends on a class/interface `B` if it extends (or implements) `B` or it has a field, a local declaration, a parameter of type `B` or it contains a cast expression to `B` or it depends on a class/interface `C` which depends on `B`.

Second, Cibai performs some syntactic transformations on the abstract syntax tree. We can divide those transformations into two classes. The first one rewrites some constructs so to reduce the number of syntactic constructs the analysis must handle. For instance in this phase Cibai rewrites `for` loops and `do...while` loops in terms of `while` loops. The second syntactic transformation instruments the code with assertions. For instance, for each array access `a[E]` we emit the two assertions `assert 0` $\leq$ `E` and `assert E` $<$ `a.length`.

Third, it analyzes the compilation unit according to such dependencies: if a class `A` depends on a class `B`, then `B` is analyzed before `A`. It may be the case

that two or more classes are mutually dependent. In its current version, Cibai breaks the dependencies by assuming the worst case. For instance, suppose that `A` depends on `B` and `B` depends on `A`. Then, when analyzing `A` we assume `B` to be unknown (*i.e.* we assume its semantics to be "top") and *vice versa*. This is quite a rough approximation. We may solve it by using a global fixpoint computation involving both `A` and `B`. Nevertheless, we plan to do it another way, namely by using the technique that we introduced in [14], which allows to split the analyses of both `A` and `B` yet preserving a good precision.

Finally, the analysis of a class boils down to the computation of (1), when instantiated with the abstract domain of Proposition 2. More precisely, as the abstract domain $\bar{\Sigma}$ does not respect the ACC, we use a widening operator to ensure the convergence of the analysis, so that we actually compute a post-fixpoint of (1). In order to improve the precision, we also use a narrowing operator. To sum up, first we apply the iteration schema

$$\begin{aligned} I^0 &= \bar{\mathbb{I}}[\![\texttt{init}]\!] \\ I^{k+1} &= I^k \bar{\sqcup} \bar{\bigsqcup}_{\texttt{m}\in\texttt{M}} \bar{\mathbb{M}}[\![\texttt{m}]\!](I^k) \bar{\sqcup} \overline{\text{Context}}(I^k) && 0 \le k \le w \\ I^{k+1} &= I^k \bar{\nabla} \left(\bar{\bigsqcup}_{\texttt{m}\in\texttt{M}} \bar{\mathbb{M}}[\![\texttt{m}]\!](I^k) \bar{\sqcup} \overline{\text{Context}}(I^k) \right) && w < k, \end{aligned}$$

to get a post-fixpoint $I^{\bar{\nabla}}$. Finite convergence to $I^{\bar{\nabla}}$ follows by the properties of the widening $\bar{\nabla}$, [6]. Next, we refine $I^{\bar{\nabla}}$ by a downward iteration:

$$I^\omega = I^0 \bar{\sqcup} \left(I^{\bar{\nabla}} \bar{\sqcap} \left(\bar{\bigsqcup}_{\texttt{m}\in\texttt{M}} \bar{\mathbb{M}}[\![\texttt{m}]\!](I^{\bar{\nabla}}) \bar{\sqcup} \overline{\text{Context}}(I^{\bar{\nabla}}) \right) \right). \tag{3}$$

Proposition 3 (Soundness of Cibai). *Under the hypotheses of Proposition 1, let I^{lfp} be the solution of (1) and I^ω be as in (3). Then $I^{\mathsf{lfp}} \bar{\sqsubseteq}_{\bar{\Sigma}} I^\omega$.*

4.2 Dynamic Octagon Operations

We were left by previous sections to the definition of the abstract operations on dynamic octagons.

Without loss of generality from now on we assume that for all the dynamic octagons, the same address corresponds to the same identifier in the octagon. Formally, we assume that

$$\forall \langle f_1, \mathsf{o}_1 \rangle, \langle f_2, \mathsf{o}_2 \rangle \in \mathsf{DynOctagon}.\ \forall \bar{\mathsf{a}} \in \text{dom}(f_1) \cap \text{dom}(f_2).\ f_1(\bar{\mathsf{a}}) = f_2(\bar{\mathsf{a}}).$$

Our implementation satisfies the such an assumption.

Join. The join of the dynamic octagons, as well as the other operations, must take into account the fact that the dynamic octagons to join may have different

dimensions. This is the case when we have the allocation of an object containing integer fields in one of the branches of a conditional.

Let $\langle f_1, \mathsf{o}_1\rangle$ and $\langle f_2, \mathsf{o}_2\rangle$ be in $\mathsf{DynOctagon}$. The *kernel* is defined as $\mathrm{dom}(f_1) \cap \mathrm{dom}(f_2)$. Let $\bar{\mathsf{d}}_1$ and $\bar{\mathsf{d}}_2$ be two dynamic octagons of size n and m. If $n == m$, then we can apply the standard join on octagons. If $n < m$, we construct the dynamic octagon $\bar{\mathsf{d}}_1 \bar{\sqcup}_{\mathsf{do}} \bar{\mathsf{d}}_2$ by first joining the octagon constraints corresponding to their kernel. Then we add the constraints involving the abstract addresses not in the kernel, *i.e.* the addresses that correspond to fresh allocated memory. In fact, if an address is in the dynamic octagon $\bar{\mathsf{d}}_1$ but not in the kernel, it means that its value in $\bar{\mathsf{d}}_2$ is bottom. As a consequence joining a constraint with the bottom value is equivalent to the constraint itself. In our case, this is equivalent to just adding the constraint to the result.

Example 1. Let $\langle f_1, o_1\rangle$ and $\langle f_2, o_2\rangle$ be two dynamic octagons such that $f_1 = \langle \bar{\mathsf{a}}_1 \mapsto \mathsf{d}_1, \bar{\mathsf{a}}_2 \mapsto \mathsf{d}_2, \bar{\mathsf{a}}_3 \mapsto \mathsf{d}_3\rangle$, $o_1 = \{\mathsf{d}_1 = 5, \mathsf{d}_1 - \mathsf{d}_2 \leq 0, \mathsf{d}_3 = 99\}$, $f_2 = \langle \bar{\mathsf{a}}_1 \mapsto \mathsf{d}_1, \bar{\mathsf{a}}_2 \mapsto \mathsf{d}_2\rangle$ and $o_2 = \{\mathsf{d}_1 = -6\}$.

The kernel is $\{\bar{\mathsf{a}}_1, \bar{\mathsf{a}}_2\}$. The address $\bar{\mathsf{a}}_2$, corresponding to the dimension d_2 is unconstrained in o_2. As a consequence it can assume any value, so that the join of the octagons projected on the kernel is $-6 \leq \mathsf{d}_1 \leq 5$.

On the other hand, the address $\bar{\mathsf{a}}_3$ is not in the kernel, *i.e.* it is not defined for the first octagon. Intuitively, it is as its value is $\bot$, so that the join is simply the constraint $\mathsf{d}_3 = 99$.

Finally, the join of the two dynamic octagons is $\langle\{\bar{\mathsf{a}}_1 \mapsto \mathsf{d}_1, \bar{\mathsf{a}}_2 \mapsto \mathsf{d}_2, \bar{\mathsf{a}}_3 \mapsto \mathsf{d}_3\}, \{-6 \leq \mathsf{d}_1 \leq 5, \mathsf{d}_3 = 99\}\rangle$. □

Definition 7 (Join of dynamic octagons, $\bar{\sqcup}_{\mathsf{do}}$). *Let* $\bar{\mathsf{d}}_1 \in \mathsf{DynOctagon}_n$ *and* $\bar{\mathsf{d}}_2 \in \mathsf{DynOctagon}_m$ *be dynamic octagons different from* $\bar{\bot}_{\mathsf{do}}$ *and* $\bar{\top}_{\mathsf{do}}$. *Let* k *be the size of the kernel of* $\bar{\mathsf{d}}_1$ *and* $\bar{\mathsf{d}}_2$. *Then their join* $\bar{\sqcup}_{\mathsf{do}} \in [\mathsf{DynOctagon}_n \times \mathsf{DynOctagon}_m \to \mathsf{DynOctagon}_{n+m-k}]$ *is defined as in Fig. 2.*

Please note that in the definition of $\bar{\sqcup}_{\mathsf{do}}$ we pay attention that for the constraints involving addresses not in the kernel we apply a renaming so to avoid (potential erroneous) overlapping of identifiers.

Meet. The meet of dynamic octagon is similar in spirit to the join. We identify the common constraints to both the dynamic octagons, and we "meet" them by using the standard meet operation on octagons, $\bar{\sqcap}_{\mathsf{o}}$. We discard all the addresses (and the corresponding constraints) not in the kernel. This is equivalent to setting the value of the addresses not in the kernel to bottom.

Definition 8 (Meet of dynamic octagons, $\bar{\sqcap}_{\mathsf{do}}$). *Let* $\bar{\mathsf{d}}_1 \in \mathsf{DynOctagon}_n$ *and* $\bar{\mathsf{d}}_2 \in \mathsf{DynOctagon}_m$ *be two dynamic octagons different from top and bottom. Let* r *the size of the kernel of* $\bar{\mathsf{d}}_1$ *and* $\bar{\mathsf{d}}_2$. *Their meet* $\bar{\sqcap}_{\mathsf{do}} \in [\mathsf{DynOctagon}_n \times \mathsf{DynOctagon}_m \to \mathsf{DynOctagon}_r]$ *is defined in Fig. 3.*

Please note that the result of the meet of two dynamic octagons is, in general, a dynamic octagon with fewer dimensions than the operands.

$$
\begin{aligned}
\bar{\mathsf{d}}_1 \bar{\sqcup}_{\mathsf{do}} \bar{\mathsf{d}}_2 = \text{let } & \bar{\mathsf{d}}_1 = \langle f_1, \mathsf{o}_1 \rangle, \bar{\mathsf{d}}_2 = \langle f_2, \mathsf{o}_2 \rangle \\
\text{in let } & \iota = \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) \\
\text{in let } & \kappa_1 = \pi_{f_1(\iota)}(\mathsf{o}_1), \kappa_2 = \pi_{f_2(\iota)}(\mathsf{o}_2) \\
\text{in let } & \eta_1 = \mathrm{dom}(f_1) - \iota, \eta_2 = \mathrm{dom}(f_2) - \iota \\
\text{in let } & \theta \text{ be a renaming from } f_1(\eta_1) \cup f_2(\eta_2) \text{ to fresh identifiers} \\
\text{in let } & \rho_1 = \theta(\pi_{f_1(\eta_1)}(\mathsf{o}_1)), \rho_2 = \theta(\pi_{f_2(\eta_2)}(\mathsf{o}_2)) \\
\text{in let } & g = \{\bar{\mathsf{a}} \mapsto i \mid \bar{\mathsf{a}} \in \iota, i = f_1(\bar{\mathsf{a}})\} \cup \{\bar{\mathsf{a}} \mapsto i \mid \bar{\mathsf{a}} \in \eta_1, i = \theta(f_1(\bar{\mathsf{a}}))\} \\
& \qquad \cup \{\bar{\mathsf{a}} \mapsto i \mid \bar{\mathsf{a}} \in \eta_2, i = \theta(f_2(\bar{\mathsf{a}}))\} \\
& \text{in} \langle (\kappa_1 \bar{\sqcup}_o \kappa_2) \cup \rho_1 \cup \rho_2, g \rangle
\end{aligned}
$$

Fig. 2. The definition of the join of dynamic octagons. $\pi_{Id}(\mathsf{o})$ is the projection of the octagon o on the identifiers Id. With an abuse of notation we lift function application to sets.

$$
\begin{aligned}
\bar{\mathsf{d}}_1 \bar{\sqcap}_{\mathsf{do}} \bar{\mathsf{d}}_2 = \text{let } & \bar{\mathsf{d}}_1 = \langle f_1, \mathsf{o}_1 \rangle, \bar{\mathsf{d}}_2 = \langle f_2, \mathsf{o}_2 \rangle \\
\text{in let } & \iota = \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) \\
\text{in let } & \kappa_1 = \pi_{f_1(\iota)}(\mathsf{o}_1), \kappa_2 = \pi_{f_2(\iota)}(\mathsf{o}_2) \\
& \text{in} \langle \kappa_1 \bar{\sqcap}_o \kappa_2, f_1 \cap f_2 \rangle.
\end{aligned}
$$

Fig. 3. The definition of the meet of two dynamic octagons. $\pi_{Id}(\mathsf{o})$ is the projection of the octagon o on the identifiers Id. The intersection of functions is defined as the intersection of the domains and the co-domains.

Widening. When performing the widening of two dynamic octagons, we identify the addresses that are common to the two. Then we perform the widening of just the constraints involving addresses in the kernel. For all the addresses not in the kernel, we keep them in the resulting dynamic octagon unconstrained. This is equivalent to setting their value to top.

Proposition 4 (Widening of dynamic octagons, $\bar{\nabla}_{\mathsf{do}}$). *Let* $\bar{\mathsf{d}}_1 \in \mathsf{DynOctagon}_n$ *and* $\bar{\mathsf{d}}_2 \in \mathsf{DynOctagon}_m$ *be two dynamic octagons different from top and bottom. Let* k *be the size of the kernel of* $\bar{\mathsf{d}}_1$ *and* $\bar{\mathsf{d}}_2$*. Let us consider the operator* $\bar{\nabla}_{\mathsf{do}}$ *in Fig. 4. If the set of abstract addresses is bounded, then* $\bar{\nabla}_{\mathsf{do}}$ *is a widening operator such that* $[\mathsf{DynOctagon}_n \times \mathsf{DynOctagon}_m \to \mathsf{DynOctagon}_{n+m-k}]$.

The proposition above requires that the set of abstract addresses to be bounded in order to have a widening operator on dynamic octagons. In our implementation we pay attention to generate finitely many abstract addresses during the analysis.

$$\begin{aligned}\bar{\mathsf{d}}_1 \bar{\nabla}_{\mathsf{do}} \bar{\mathsf{d}}_2 = \text{let } & \bar{\mathsf{d}}_1 = \langle f_1, \mathsf{o}_1 \rangle, \bar{\mathsf{d}}_2 = \langle f_2, \mathsf{o}_2 \rangle \\ \text{in let } & \iota = \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) \\ \text{in let } & \kappa_1 = \pi_{f_1(\iota)}(\mathsf{o}_1), \kappa_2 = \pi_{f_2(\iota)}(\mathsf{o}_2) \\ & \text{in} \langle \kappa_1 \bar{\nabla}_{\mathsf{o}} \kappa_2, f_1 \cup f_2 \rangle.\end{aligned}$$

Fig. 4. The definition of the widening of two dynamic octagons. $\pi_{Id}(\mathsf{o})$ is the projection of the octagon o on the identifiers Id. The union of functions is defined as the union of the domains and of the co-domains.

4.3 Transfer Functions

The analysis of the bodies of the constructors and the methods is by induction on the syntax of the program. The analyzer provides transfer functions for most of the constructors of sequential Java. It does not support reflection. Here we describe the implementation of the most interesting transfer functions, the others being quite standard.

Parameters. Before analyzing the body of a constructor or of a method we have to set up the initial state considering the input parameters. We describe the parameter initialization just for methods, the constructor's case being easier. We distinguish two cases, whether the parameters are of basic type or of a reference type.

If the method input parameters are of a basic type, we know that no aliasing may be created thanks to the semantics of parameter passing in Java. We extend the abstract environment with the new variable and assume the value to be unknown.

Example 2. Let $\langle \bar{\mathsf{e}}, \langle \bar{\mathsf{b}}, \langle f, \mathsf{o} \rangle, \bar{\mathsf{r}} \rangle, S \rangle$ be an abstract state. Let `void m(bool b, int i)` $\{\ldots\}$ be a public method. Let $\bar{\mathsf{a}}_1$ and $\bar{\mathsf{a}}_2$ be fresh addresses, id be a fresh identifier, and o' be the octagon o extended with a new dimension corresponding to the identifier id. Then the initial abstract state for the analysis of the body of `m` is $\langle \bar{\mathsf{e}}[\mathsf{b} \mapsto \{\bar{\mathsf{a}}_1\}, \mathsf{i} \mapsto \{\bar{\mathsf{a}}_2\}], \langle \bar{\mathsf{b}}[\bar{\mathsf{a}}_2 \mapsto \bar{\top}], \langle f[\bar{\mathsf{a}}_1 \mapsto \mathsf{id}], \mathsf{o}' \rangle, \bar{\mathsf{r}} \rangle, S \rangle$. □

If the parameters are of reference type we have to pay attention to possible aliasing. We assume that parameters of the same type (or subtype) may alias. For instance if we have

```
class A { B bRef; }
class B { int i; }
class ToAnalyze {
//...
public void m(A a, B b) { ... } }
```

Then `a.bRef` and `b` may alias. Our choice is to use summary locations for objects of a type that appears at least twice in the method's parameters.

Example 3. Referring to the classes `A`, `B` and `ToAnalyze` above, let us consider an input abstract state $\langle \bar{e}, \langle \bar{b}, \langle f, o\rangle, \bar{r}\rangle, S\rangle$ for the analysis of `m`. Let $\bar{a}_1$, be an abstract address, $\bar{a}_2^*$, $\bar{a}_3^*$ be summary abstract addresses, `id` a fresh identifier and o' the octagon `o` extended with the new identifier `id`. Furthermore let $\bar{e}' = \bar{e}[a \mapsto \{\bar{a}_1\}, b \mapsto \{\bar{a}_2^*\}]$, $f' = f[\bar{a}_2^* \mapsto id]$, $\bar{r}' = \bar{r}[\bar{a} \mapsto \langle\{A\}, [bRef \mapsto \bar{a}_1^*]\rangle, \bar{a}_1^* \mapsto \langle\{B\}, [i \mapsto \bar{a}_2^*]\rangle]$, then the initial state for the analysis of the body of `m` is $\langle \bar{e}[a \mapsto \{\bar{a}_1\}, b \mapsto \{\bar{a}_1^*\}], \langle \bar{b}, \langle f', o'\rangle, \bar{r}'\rangle, S\rangle$. □

Assignments. When performing an assignment $e_1 = e_2$ we distinguish between two cases depending whether the static type of e_1 is a basic type or a reference type. Furthermore, there is the orthogonal issue of considering if we are assigning to a location corresponding to a summary address. We skip here the technical details of our implementation, but the intuition behind is that (i) the update of a location corresponding to an *exact* abstract address is destructive, in that the old value is replaced by the new one; and (ii) the update of a location corresponding to a *summary* abstract address is *weak*, in that the new value is joined with the old one.

The assignment to a boolean consists in (i) the evaluation of e_2; and (ii) the update of the abstract location corresponding to e_1.

The assignment to an integer is more complicated as e_1 and e_2 may evaluate to several addresses, and we want to keep a relational information between the `int`-locations.

Example 4. Suppose to have the assignment $a.x = b.c.y + 2$ in an abstract state $\langle \bar{e}, \langle \bar{b}, \langle f, o\rangle, \bar{r}\rangle, S\rangle$ where, for some classes `C`, `D`, `E`:

$$\begin{aligned}
\bar{e} =& [a \mapsto \{\bar{a}_1, \bar{a}_2\}, b \mapsto \{\bar{a}_3\}] \\
f =& [\bar{a}_4 \mapsto id_0, \bar{a}_5 \mapsto id_1, \bar{a}_7 \mapsto id_2] \\
\bar{r} =& [\bar{a}_1 \mapsto \langle\{C\}, x \mapsto \{\bar{a}_4\}\rangle, \bar{a}_2 \mapsto \langle\{C\}, x \mapsto \{\bar{a}_5\}\rangle, \\
& \bar{a}_3 \mapsto \langle\{D\}, c \mapsto \{\bar{a}_6\}\rangle, \bar{a}_6 \mapsto \langle\{E\}, y \mapsto \{\bar{a}_7\}\rangle].
\end{aligned}$$

Resolving the variable addresses using this abstract state, we get the possible assignments $\bar{a}_4 = \bar{a}_7 + 2$ and $\bar{a}_5 = \bar{a}_7 + 2$. The corresponding octagon constraints are $id_0 = id_2 + 2$ and $id_1 = id_2 + 2$. We use the assignment on octagons to get two new octagons $o_1 = o.\text{assign}(id_0 = id_2 + 2)$ and $o_2 = o.\text{assign}(id_1 = id_2 + 2)$. The octagon after the assignment is the join of the two cases : $o' = o_1 \bar{\sqcup}_o o_1$. □

To sum up, when analyzing an assignment to an `int`-location, we first consider all the octagons constraints that are enabled by the incoming abstract state, then we create enough copies of the incoming octagon, perform the assignments independently, *i.e.* a constraint for each duplicated octagon, and we join all the octagons together.

Finally, if in the assignment $e_1 = e_2$ the static type of e_1 is of a reference type, then the effect of the assignment is to create an alias for e_2. Therefore, we update the abstract environment (and the abstract store if needed) to reflect the fact that e_1 points to the same abstract addresses e_2 points to.

Assertions. We have two kinds of assertions: (i) those explicitly written by the programmer; and (ii) those generated by the program-transformation phase. The last ones include array access bounds-checking and division by zero. Our choice is to handle the two kind of assertions in the same way: During the analysis, when we reach an assertion (i) we check whether the assertion does hold, does not hold, or "*we do not know*" in the incoming abstract state; and (ii) we meet the incoming state with the asserted expression so to produce the outgoing abstract state.

Object Instantiations. Objects are created through the `new` statement. In the concrete, the invocation of `new` returns an address to a freshly allocated memory location where to store the object. Such an address is the identity of the object. In the abstract, in order to guarantee the convergence of the analysis we limit, for each `new` statement in the source code, the allocation of just k fresh objects. k is a command line parameter of the analyzer. After k exact instantiations, we create a summary location that collects the behavior of all the other objects that can be instantiated in such particular statement.

Return statements. We join together all the abstract states that reach a `return` statement in the body of a method. Let $\langle \bar{\mathsf{e}}, \langle \bar{\mathsf{b}}, \langle f, \mathsf{o} \rangle, \bar{\mathsf{r}} \rangle, S \rangle$ the abstract state at the method's exit point. We assume that the returned value is stored into a special variable `#ret`.

If `#ret` is of a basic type, then no address reachable from one of the fields of the class we are analyzing is exposed to the context, [15]. Then we do not need to add any new address to S, the set of exposed abstract addresses. Please note that the statements that constitute the body of the method do not affect either S, too.

If `#ret` is of a reference type, then it is possible that the returned object may contain, in its fields, references to memory locations that are reachable also from the fields of the class we are analyzing. Therefore, we determine the sets of abstract addresses reachable from the fields of the class under analysis, F, and then of addresses reachable from the returned reference, R. The abstract addresses in $F \cap R$ may be exposed to the context, so we update S to be $S \cup (F \cap R)$. In Cibai, we perform a similar reasoning also for parameters.

Abstract context. The abstract context over-approximate the behavior of a worst-context, *i.e.* a context that once is aware of an address it changes its value arbitrarily. The input to the $\overline{\text{Context}}$ is an abstract state $\langle \bar{\mathsf{e}}, \langle \bar{\mathsf{b}}, \langle f, \mathsf{o} \rangle, \bar{\mathsf{r}} \rangle, S \rangle$. Then, for each abstract address in S, it sets the corresponding value to top.

5 Experiments

We have compared Cibai and ESC/Java 2 on a class library consisting of 2800 lines of sequential Java code. We did not annotated nor modified the code. The library is quite representative in that its code contains dynamic memory

allocation, non-trivial loops, method calls and aliasing issues. Because of its nature, it also allows the testing of the modular aspects of the tools.

The testing platform is a 1.40 GHz Pentium M laptop, with 632MB of RAM, running Windows XP Service Pack 2 and the Sun JVM version 1.6 beta 2. Cibai analyzed the 40 files of the library in 28.6 seconds, whereas ESC/Java took 50.2 seconds. Cibai was able to verify the library except for two assertions involving non-octogonal arithmetic relations. ESC/Java 2 emitted 107 warnings, mainly about null dereferences and out of bound array accesses.

In order to understand the reasons of the difference in precision, let us consider the initial example (cf. Fig. 1). ESC/Java is not able to infer that (i) the field `elements` is always non-null, as it is allocated in the constructor and never modified; (ii) the field `top` is always positive and smaller or equal to the length of the array. As a consequence, it cannot check the array accesses and dereferences in the body of the methods of `BagOfInts`, so it emits the warnings. To overcome this problem, the tool requires the user to modify the code by adding annotations. On the other hand, Cibai can infer the right class invariants for `BagOfInts`, so that without any user interaction it can verify the class to be correct.

We have tuned the performances by using a profiler to find the bottlenecks. For instance, in one of the test cases, written on purpose to stress the allocation of many objects containing integer fields, we experienced very bad performances. The profiler showed that the problem was because of the closure operation on octagons. Closure is used everywhere: for emptiness checks, for the order, etc. In a first version, octagons were represented as a square matrix of `IExtendedInt`. `IExtendedInt` is an interface implemented by three classes `AnInt`, `PlusInf` and `MinusInf`. We replaced the representation of octagons with a square matrix of `int`, `maxint` and `minint` standing respectively for $+\infty$ and $-\infty$. We also implemented a form of sharing of octagons. With the optimized representation, we obtained an improvement of the 900% of the performances in the stress test. We conjecture that such a dramatic speedup is obtained because the matrix of integers can be stored inside the processor cache, so that each access is internal to the processor.

6 Related Work

There are several tools for the static analysis and verification of object-oriented languages. Most of them are based on the Java Modeling Language (JML), [13]. JML is a Hoare-like logic for the specification of Java program.

ESC/Java 2 tries to check that a JML-annotated program satisfies its specification. It is neither sound nor complete, [5]. Other tools as Jack [23], LOOP [12], Krakatoa [17] are sound, but they present a low level of automation. In fact, they need the programmer to supply, among the other others, loop invariants. Furthermore, as they are based on interactive theorem provers, they also need the user assistance for conducting the proof.

Spec# adds the support for design-by-contract to C#, [3]. Contracts are checked either dynamically or statically. The Spec# static program verifier,

Boogie, integrates an inference engine for reduce the burden of program annotations when dealing with loops. Nevertheless, the inference engine inside Boogie is quite limited and it is not able to infer class invariants.

Daikon is a tool for discovering class invariants-like in Java, [8]. It monitors the execution of a program trying to determine if some simple properties hold or not. Being based on testing, it is inherently unsound. Axiom Meister is a step forward w.r.t. Daikon as it tries to infer preconditions in `.net` programs using symbolic execution, [24]. The discover of algebraic specifications for Java classes is also the goal of the tool introduced in [11].

Julia is generic static analyzer that works on the bytecode level, [22], and it focuses mainly on non-numeric and non-relational properties. Jail is a static analyzer specialized for the verification of applet isolation in JavaCard, [9]. Other static analyses for object-oriented languages infer particular properties as escape analysis [4], shape analysis [20], data structures cyclicity [21] or the inference of very simple properties on class fields [2].

7 Conclusions and Future Work

We described Cibai, an automatic tool for the analysis of Java programs. The tool can analyze classes in isolation, can infer class invariants and method postconditions. It uses such information to prove the absence of some runtime errors, as the violation of user-defined assertions, the access out of the bounds of arrays or the dereference of null objects. The abstract domain underlying Cibai is a composition of an alias analysis, so to precisely track the identity of objects, and several other domains to approximate the values of Java values, as `int`s, `boolean`s or references. The structure of the analyzer is modular, so that is possible to change the approximation for one class of values, without having to modify the others.

The next step in the development of Cibai is the generation of method summaries: we plan to apply Cibai on the Java API so to generate stubs, and then to reuse these stubs when analyzing large programs. We want also to improve the handling of mutually recursive classes, using the technique we introduced in [14]; and we want to provide a translation of the inferred invariants in JML, so to automatically produce code annotations.

References

1. JUnit. `http://junit.sourceforge.net/`.
2. A. Aggarwal and K. H. Randall. Related field analysis. In *PLDI '01*.
3. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*.
4. B. Blanchet. Escape Analysis: Correctness proof, implementation and experimental results. In *POPL'98*.
5. D. R. Cok and J. Kiniry. ESC/Java 2: Uniting ESC/Java and JML. In *CASSIS 2004*.

6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*.
7. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4), August 1992.
8. M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.
9. P. Ferrara. JAIL: Firewall analysis of JavaCard by Abstract Interpretation. In *EAAI 2006*.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification - 2nd Edition*. Sun Microsystems, 2001.
11. J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *ECOOP'03*.
12. B. Jacobs, J. van den Berg, H. Huismann, M. van Berkum, U. Hensel, and Tews H. Reasoning about Java classes (preliminary report). In *OOPSLA'98*.
13. G. T. Leavens, A. L. Baker, and C. Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*, November 2003.
14. F. Logozzo. Separate compositional analysis of class-based object-oriented languages. In *AMAST'2004*.
15. F. Logozzo. *Modular Static Analysis of Object-oriented languges*. PhD thesis, École Polytecnique, 2004.
16. F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems and Structures*, 2007.
17. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certificationof Java/Javacard programs annotated in JML. *J. Log. Algebr. Program*, 58(1–2), 2004.
18. B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.
19. A. Miné. The octagon abstract domain. In *AST 2001*.
20. I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and sharing domains for static analysis of Java programs. In *ECOOP '01*.
21. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *VMCAI'06*.
22. F. Spoto. Julia: A generic static analyser for the java bytecode. In *FTfJP'05*.
23. Everest Team. Jack, Java Applet Correctness Kit. `http://www-sop.inria.fr/everest/soft/Jack/jack.html`.
24. N. Tillmann, F. Chen, and Schulte. W. Discovering likely method specifications. Technical report, Microsoft Research, 2006.