Analyzing Java programs with Octagons

September 20

Francesco Logozzo

ENS

Goals

We want a static analysis for sequential Java that

- Is sound
  - $\neq$ ESC/Java, ESC/Java 2
- Is automatic
  - $\neq$ Jive
- Infers real invariants
  - $\neq$ Daikon
- Analyze classes in isolation
  - $\neq$ Excelsion Flawdetector
- Does not rely on user annotations
  - $\neq$ JML-based approaches as LOOP, Krakatoa

Outline

- Concrete Semantics
- Abstract Semantics
- Comparison with other tools
- Conclusions & Future work

Syntax

- A class is a triplet $\langle \texttt{init}, \texttt{F}, \texttt{M} \rangle$ where
  - `init` is the class constructor
  - `F` is a set of variables
  - `M` is a set of function definitions
- For simplicity assume
  - Just one constructor
  - All the fields are protected
    - Otherwise use `get_f`/`set_f`

Semantic domains

- The set of **environments** is $\mathsf{Env} = [\mathtt{Var} \to \mathsf{Addr}]$ where
  - $\mathtt{Var}$ is a set of variables
  - Addr is a set of addresses
- The the set of **stores** is $\mathsf{Store} = [\mathsf{Addr} \to \mathsf{Val}]$ where
  - Val is a set of values such that $\mathsf{Env} \subseteq \mathsf{Val}$
  - $\phi \in \mathsf{Val}$ is the void value
- The environment of an object is stored in a certain memory location
- The address of such a location is the object identity
- A program state is $\sigma \in \mathsf{Env} \times \mathsf{Store}$

Constructor and method semantics

- The semantics of the constructor is a function

$$\mathbb{i}[\![\texttt{init}]\!] \in [\mathsf{Val} \times \mathsf{Store} \to \mathcal{P}(\mathsf{Env} \times \mathsf{Store})]$$

- The semantics of a method $\texttt{m}$ is a function

$$\mathbb{m}[\![\texttt{m}]\!] \in [\mathsf{Val} \times \mathsf{Env} \times \mathsf{Store} \to \mathcal{P}(\mathsf{Val} \times \mathsf{Env} \times \mathsf{Store})]$$

- If $\mathsf{Addr} \subseteq \mathsf{Val}$ then the method may expose a part of object state to the context

Object semantics

- The semantics of an object is given by a set of traces
  - Each trace represents a possible evolution history of the object
- The first state represents the object right after its creation
- Each further state is the result of an interaction between the object and a context
  - The context invokes a method of the object; or
  - It modifies a memory location that is reachable from the object environment
- The semantics of a class considers all its instances

Initial states

- The set of interaction states is

$$\Sigma = \mathsf{Env} \times \mathsf{Store} \times \mathsf{Val} \times \mathcal{P}(\mathsf{Addr})$$

- The initial states of an object $\mathtt{o}$ is

$$S_0\langle \mathsf{v}, \mathsf{s}\rangle = \{\langle \mathsf{e}', \mathsf{s}', \phi, \emptyset\rangle \mid \langle \mathsf{e}', \mathsf{s}'\rangle \in \mathbb{i}[\![\mathtt{init}]\!](\mathsf{v}, \mathsf{s})\}$$

- The initial states of a class are

$$\mathbb{I}[\![\mathtt{init}]\!] = \cup\{S_0\langle \mathsf{v}, \mathsf{s}\rangle \mid \langle \mathsf{v}, \mathsf{s}\rangle \in \mathsf{Val} \times \mathsf{Store}\}$$

Direct interaction

- Let $\text{reachable} \in [(\mathsf{Val} \times \mathsf{Store}) \to \mathcal{P}(\mathsf{Addr})]$
  - It determines all the memory addresses reachable from an address in Val
- The collecting semantics of a method $\mathtt{m}$ is

$$\mathbb{M}[\![\mathtt{m}]\!](S) = \{\langle \mathsf{e}', \mathsf{s}', \mathsf{v}', \mathsf{Esc}' \rangle \mid \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc} \rangle \in S, \mathsf{v}_{\mathsf{in}} \in \mathsf{Val}, \\ \langle \mathsf{v}', \mathsf{e}', \mathsf{s}' \rangle \in \mathbb{m}[\![\mathtt{m}]\!](\mathsf{v}_{\mathsf{in}}, \mathsf{e}, \mathsf{s}), \\ \mathsf{Esc}' = \mathsf{Esc} \cup \text{reachable}(\mathsf{v}', \mathsf{s}')\}$$

- The transition function for the direct interaction is

$$\text{next}_{\text{dir}}(\sigma) = \cup\{\mathbb{M}[\![\mathtt{m}]\!](\{\sigma\}) \mid \mathtt{m} \in \mathtt{M}\}$$

Indirect interaction

- Let $\mathrm{update} \in [\mathsf{Addr} \times \mathsf{Store} \to \mathcal{P}(\mathsf{Store})]$ be defined as

$$\mathrm{update}(\mathsf{a}, \mathsf{s}) = \{\mathsf{s}' \mid \exists \mathsf{v} \in \mathsf{Val}.\ \mathsf{s}' = \mathsf{s}[\mathsf{a} \mapsto \mathsf{v}]\}$$

- The context interactions are

$$\mathrm{Context}(S) = \{\langle \mathsf{e}, \mathsf{s}', \phi, \mathsf{Esc}\rangle \mid \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle \in S, \\ \exists \mathsf{a} \in \mathsf{Esc}.\ \mathsf{s}' \in \mathrm{update}(\mathsf{a}, \mathsf{s})\}$$

- The transition function for the indirect interaction is

$$\mathrm{next}_{\mathrm{ind}}(\sigma) = \mathrm{Context}(\{\sigma\})$$

Class semantics, $\mathbb{c}[\![\cdot]\!]$

- Th. $\mathbb{c}[\![\mathtt{A}]\!]$ can be expressed in fixpoint form as

$$\mathbb{c}[\![\mathtt{A}]\!] = \mathrm{lfp}\lambda T.\ \mathbb{I}[\![\mathtt{init}]\!] \cup \{\sigma_0 \to \ldots \to \sigma_n \to \sigma' \mid \sigma_0 \to \ldots \to \sigma_n \in T, \sigma' \in \mathrm{next}_{\mathrm{dir}}(\sigma_n) \cup \mathrm{next}_{\mathrm{ind}}(\sigma_n)\}$$

- $\mathbb{c}[\![\mathtt{A}]\!]$ can be shown sound and complete w.r.t. a whole program object-oriented semantics
- The class reachable states of a class $\mathtt{A}$ are

$$\mathbb{C}[\![\mathtt{A}]\!] = \mathrm{lfp}\lambda S. \mathbb{I}[\![\mathtt{init}]\!] \cup \bigcup_{\mathtt{m} \in \mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup \mathrm{Context}(S)$$

Outline

- Concrete Semantics
- Abstract Semantics
- Comparison with other tools
- Conclusions & Future work

Abstraction of the reachable states

- Assume an abstract domain $\bar{\mathsf{D}}$ such that

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$$

- A sound approximation of the initial states

$$\mathbb{I}[\![\texttt{init}]\!] \subseteq \gamma(\bar{\mathbb{I}}[\![\texttt{init}]\!])$$

- A sound approximation of the method semantics

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

- A sound approximation of the context

$$\forall \mathsf{S} \in \mathcal{P}(\Sigma).\ \mathrm{Context}(\mathsf{S}) \subseteq \gamma(\overline{\mathrm{Context}}(\alpha(\mathsf{S})))$$

Abstract class invariant

- Th. (SAS'03) Let $\mathtt{A}$ be a class. Then

$$\bar{\mathbb{C}}[\![\mathtt{A}]\!] = \mathrm{lfp}\,\lambda X.\ \bar{\mathbb{I}}[\![\mathtt{init}]\!]\bar{\sqcup}\overline{\bigsqcup_{\mathtt{m}\in\mathtt{M}}}\bar{\mathbb{M}}[\![\mathtt{m}]\!](X)\bar{\sqcup}\overline{\mathrm{Context}}(X)$$

  is such that

$$\mathbb{c}[\![\mathtt{A}]\!] \subseteq \gamma_\Sigma \circ \gamma(\bar{\mathbb{C}}[\![\mathtt{A}]\!])$$

- In practice:
  - We have to define $\bar{\mathbb{I}}[\![\cdot]\!]$ and $\bar{\mathbb{M}}[\![\cdot]\!]$
  - The abstract domain $\bar{\mathrm{D}}$ must track object aliasing
    $\Longrightarrow$ In particular the exposed objects
    $\Longrightarrow$ $\overline{\mathrm{Context}}$ sets them to $\bar{\top}$

Abstract domain: Abstract environment

- The abstract domain is $\overline{\mathsf{D}} = \overline{\mathsf{Env}} \times \overline{\mathsf{Store}}$
- An abstract environment overapproximates the set of addresses a variable may be tied to
- The abstract environments are

$$\overline{\mathsf{Env}} = [\mathtt{Var} \to \mathcal{P}(\overline{\mathsf{Addr}})]$$

- $\overline{\mathsf{Addr}}$ is the set of abstract addresses
- The meaning of a $\overline{\rho} \in \overline{\mathsf{Env}}$ is

$$\gamma(\overline{\rho}) = \{\rho \in \mathsf{Env} \mid \forall \mathtt{x}.\ \rho(\mathtt{x}) \in \gamma_{addr}(\overline{\rho}(\mathtt{x}))\}$$

Abstract domain: Abstract store

- Assume the only primitive type being `int`
- The abstract stores are

$$\overline{\mathsf{Store}} = \underbrace{([\overline{\mathsf{Addr}} \to \mathbb{N}] \times \mathsf{Oct})}_{\text{Approximation of } \texttt{int}\text{s}} \times \underbrace{[\overline{\mathsf{Addr}} \to \overline{\mathsf{Env}}]}_{\text{Approximation of objects}} \times \underbrace{\mathcal{P}(\overline{\mathsf{Addr}})}_{\text{Summary locations}}$$

- Let $\langle\langle g, o\rangle, f, S\rangle \in \overline{\mathsf{Store}}$ then
  - $dom(f) \cap dom(g) = \emptyset$
    $\Longrightarrow$ An address denotes objects or integers
    $\Longrightarrow$ Typed addresses

- S is the set of summary locations
  - if $\overline{a} \in S$ then $\overline{a}$ stands for (infinitely?) many concrete addresses
  - Otherwise $\overline{a}$ stands for exactly one concrete address
- $f \in [\overline{\mathsf{Addr}} \rightarrow \overline{\mathsf{Env}}]$ associates an abstract environment to each abstract object
- $g \in [\overline{\mathsf{Addr}} \rightarrow \mathbb{N}]$ maps `int`s to octagon's dimensions
  - A dimension is associated to each address
  - Useful in implementation
    - Max dimension not known statically

Example

```
class Box { int i = 0; }
// ...
```

`//` $\langle \rho, \langle \langle g, o \rangle, f, S \rangle \rangle$, o with n dimensions

```
Box b = null;
if( ... )
  b = new Box();
else skip;
```

`//` $\langle \rho[\mathsf{b} \mapsto \{a\}], \langle \langle g[a \mapsto (n+1)], o[x_{n+1} = 0] \rangle, f, S \rangle \rangle$

Note: At the join point we add the constraint $x_{n+1} = 0$

$\Longrightarrow$ In the false branch: $x_{n+1} = \bot$ is tacitely assumed

Abstract operations and transfer functions

- The abstract operations are defined as one expects
  - e.g., $\sqcup$, $\sqcap$, ∇ are pointwisely defined
- Most of the transfer functions are as usual
  - Sequence, `if`, `while`, ...
- Next we describe the most interesting ones
  - Set up the initial state for the methods' analysis
  - Creation of a new object (`new`)
  - Assignment (:=)
  - Projection of exposed addresses

Handling parameters

- Let $I^k = \langle \rho, \langle \langle g, o \rangle, f, S \rangle \rangle$ be the state at the k-th iteration for the class invariant
- The initial state I_0 for a method `m` is given by
  - The class invariant computed so far (I^k)
  - The parameters of the method
- Parameters can be `int`s or objects
  - If it is an `int` we simply add it to the state, e.g.:

    ```
    public void m(int v) { ... }
    ```

    then $I_0 = \langle \rho[\mathtt{v} \mapsto \overline{a}], \langle \langle g[\overline{a} \mapsto n+1], o' \rangle, f, S \rangle \rangle$ where o' is the octagon o extended to the $n+1$ dimension

- If they are objects, we have to pay attention to
  - aliasing, eg:

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

$\Longrightarrow$ `a.bRef` and `b` may alias

  - cycles, eg:

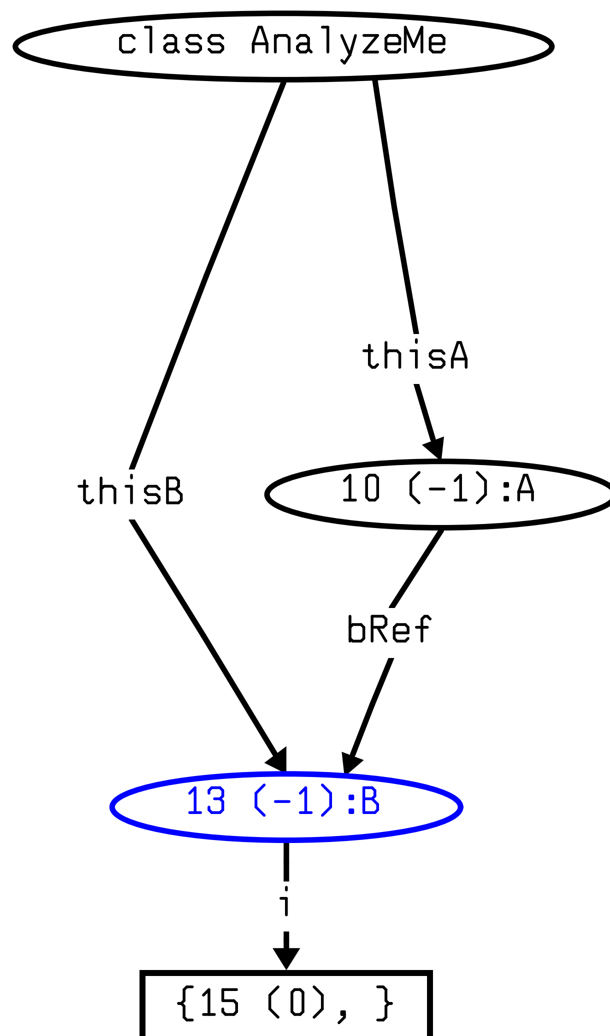
```
class Cycle_1 { Cycle_2 c1to2; // ...}
class Cycle_2 { Cycle_1 c2to1; // ...}
class AnalyzeMeWithCycles {
//
public void n(Cycle_1 c) { ... } }
```

$\Longrightarrow$ `c` and `c.c1to2.c2to1` may alias

- We use summary locations (we assume worst case) for objects of a type that appear at least twice in method's parameters
- Ex: `AnalyzeMe`: $I_0 = \langle \rho', \langle\langle g', o' \rangle, f', S' \rangle\rangle$ where
  - ρ' = $\rho[\mathtt{a} \mapsto \{13\}, \mathtt{b} \mapsto \{10\}]$
  - g' = $g[15 \mapsto (n+1)]$
  - o' = o extended with an $n+1$th dimension
  - f' = $f[13 \mapsto \langle \mathtt{i} \mapsto \{15\} \rangle, 10 \mapsto \langle \mathtt{bRef} \mapsto \{13\} \rangle]$
  - S' = $S \cup \{13, 15\}$
- (A more understandable picture will come)

Output of the Analyzer

No exceptions




```
# of dimensions: 8
# of allocated cells: 288
# of constraints: 0
# of addresses: 1 : [15 (0)]
```

`new`

- Objects are created through the `new` statement

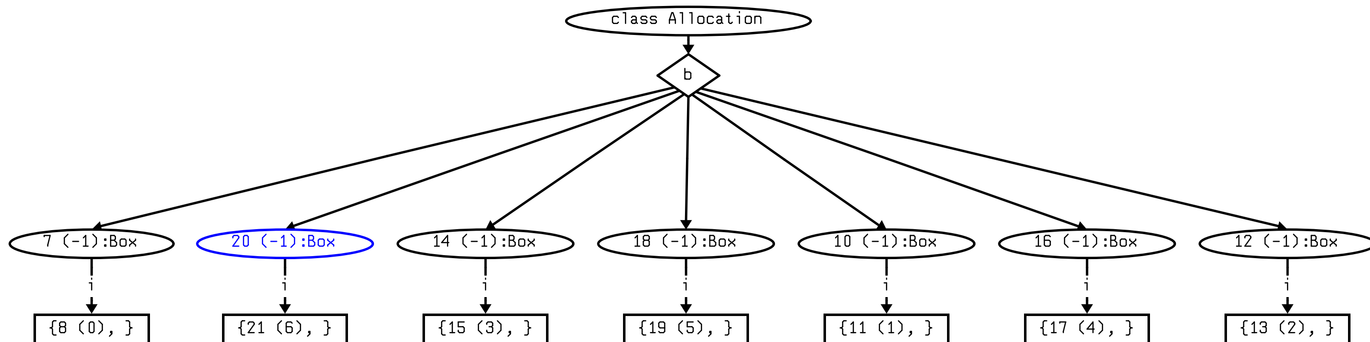
$$\texttt{A a} = \texttt{new A}();$$

- We must not to create infinitely many objects
  - Loops and successive invocations of the same method
- We handle the first k invocations of a `new` exactly and then we create a summary location
  - k is a parameter of the analysis

Example

```
class Box { int i; }

class Allocation {
    Box b;
    Allocation() { b = new Box(); }
    public void m() { b = new Box(); }
}
```



Assignment

- Let `e1` $=$ `e2` be an assignment
- Two cases: whether `e1` is an object or an `int`
  - Java is a typed language...
- If it is an object, then create an alias for `e2`
  $\Longrightarrow$ Update the abstract environment
- If it is an `int`, then we have to update the octagon

- `e1` and `e2` may evaluate to several addresses, eg:

$$\texttt{a.x} = \texttt{b.c.y} + 2$$

with, eg, $I_0 = \langle \rho, \langle \langle g, o \rangle, f, S \rangle \rangle$ and

- ρ = $\langle \texttt{a} \mapsto \{a_1, a_2\}, \texttt{b} \mapsto \{a_3\} \rangle$
- g = $\langle a_4 \mapsto d_0, a_5 \mapsto d_1, a_7 \mapsto d_2 \rangle$
- f = $\langle a_1 \mapsto \langle \texttt{x} \mapsto \{a_4\} \rangle, a_2 \mapsto \langle \texttt{x} \mapsto \{a_5\} \rangle,$
  $a_3 \mapsto \langle \texttt{c} \mapsto \{a_6\} \rangle, a_6 \mapsto \langle \texttt{c} \mapsto \{a_7\} \rangle \rangle$

The possible assignments are

$$a_4 = a_7 + 2 \text{ and } a_5 = a_7 + 2$$

that become the octagon constraints

$$d_0 = d_2 + 2 \text{ and } d_1 = d_2 + 2$$

finally, the octagon after the assignment is the union of the two possible octagons:

$$o' = o.\mathsf{assign}(d_0 = d_2 + 2) \sqcup o.\mathsf{assign}(d_1 = d_2 + 2)$$

so that the state after the assignment is

$$I_1 = \langle \rho, \langle \langle g, o' \rangle, f, S \rangle \rangle$$

Assignment of `int`s: Summary

- Let $\texttt{e1} = \texttt{e2} + c$ and $I_0 = \langle \rho, \langle \langle g, o \rangle, f, S \rangle \rangle$
- Let $\mathbb{A}[\![\texttt{e1}]\!](I_0)$ and $\mathbb{A}[\![\texttt{e2}]\!](I_0)$ the addresses `e1` (risp. `e2`) may evaluate to in I_0
- Let

$$\mathsf{Ass} = \{ g(a_1) = g(a_2) + c \mid a_1 \in \mathbb{A}[\![\texttt{e1}]\!](I_0), a_2 \in \mathbb{A}[\![\texttt{e2}]\!](I_0) \}$$

- Then the octagon after the assignment is

$$o' = \bigsqcup_{d_1 = d_2 + c \in \mathsf{Ass}} o.\mathsf{assign}(d_1 = d_2 + c)$$

Method exit point

- At method's exit point we collect the addresses that escape from the class
  - Somehow similar to the concept of ownership types
- They are reachable
  - by the method's / constructor parameters
  - by the methods retur value
- We set the corresponding location to $\top$
  - The context can do everything on an exposed location...

Outline

- Concrete Semantics
- Abstract Semantics
- Comparison with other tools
- Conclusions & Future work

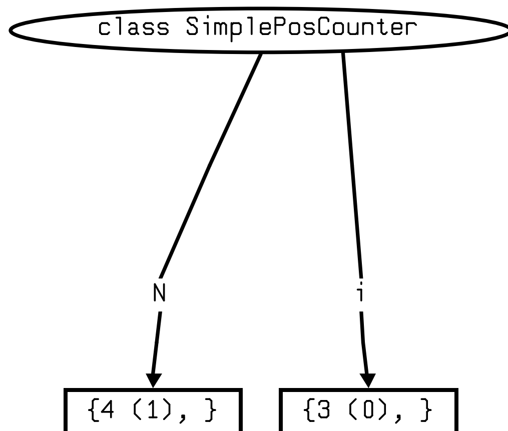
A Running example

```
class SimplePosCounter {
  private int i, N;

  SimplePosCounter(int N0) {
     assert -N0 <= -1;
     this.N = N0;
     this.i = 0; }
  public void add() {
     if(this.i - this.N < 0)
         i = i + 1; }
  public void sub() {
     if(-this.i <= this.N < 0)
         i = i - 1; }
  public void check() {
      assert N - i <= -1; }
}
```

My analyzer

- Discover that the assertion in `check()` is always violated
  - in 266ms on my laptop (Centrino 1.2Ghz, 768Mbyte, WinXp, Java 1.5_04)



No exceptions

```
# of dimensions: 8
# of allocated cells: 288
# of constraints: 4
# of addresses: 2 : [4 (1), 3 (0)]

-a3 <= 0
-a4 <= -1
a3 - a4 <= 0
-a3 - a4 <= -1
```

ESC/Java 2

- Let us consider ESC/Java 2
  - Unsound, but not for the example...
- It is made to discover errors...
- However, the wrong program passes without not even a warning...

```
< omissis >
SimplePosCounter: check() ...
    [0.016 s 6625592 bytes]  passed
```

- Is is even slower than ours: 297ms

Daikon

- It executes the program, and it infers properties from the traces
- Very popular...
- It needs that user supplies test cases
  - Unlike what suggested by the tool’s home page

$\Longrightarrow$ Cannot analyze the example directly
$\Longrightarrow$ Write the test case by myself

```
class TesterSimplePosCounter {
  public static void main(String[] unused) {

// ... Repeat 10 times
   s = new SimplePosCounter( ... );

  // ... repeat Math.random() times
  double tmp = Math.random();

  if(tmp <= 0.475)
    s.add();
  else if(tmp <= 0.999)
    s.sub();
  else
    s.check();
}
```

- If the Daikon executes `s.check()` then it stops $\Longrightarrow$ Execute it many times before success
- Finds wrong “invariants” ! eg

```
<omissis>

SimplePosCounter:::OBJECT
this.i >= 0
this.i < this.N
```

Excelsior Flawdetector

- A commercial product (free trial) to statically verify:
  - `ArrayIndexOutOfBoundsException`
  - `NullPointerException`
  - Simple assertions
  - etc.
- Now: "*Excelsior FlawDetector is temporarily unavailble due to technical issues.*"
- However, I had a trial
  - not compatible with Java 5.0
- It requires a class with `main`
  $\Longrightarrow$ Cannot analyze classes in isolation
  $\Longrightarrow$ Just issues a warning for the `check`()

JML-based/style tools

- Krakatoa, LOOP, Jack and Spec#
  - Spec# is as JML but for C#
- They need annotations
- In particular the class invariant:

$$\texttt{invariant}\ \texttt{i} <= \texttt{N}$$

  to prove the class
- In practice these tools require heavy annotations

Conclusions

- We presented the theory and some of the abstract operations of our analyzer
- There is a lot of work to be done:
  - Implementation details
    - Some optimizations, some syntax transformations, etc.
  - Modular handling of inheritance (VMCAI'04 paper)
  - Better handling of mutually recursive classes (AMAST'04 paper)
  - Does it scales up?
    - eg. Object caching, used in the implementation, does scales up?

Thank you!