Assertion-based Loop Invariant Generation
or, Counter-example Refinement Backwards

Mikoláš Janota

System Research Group,
University College Dublin, Ireland

Mobius
IST-15905
we are looking for loop invariants that will show that something “bad” will not happen

```java
for (int i = 0; i < a.length - 1; i++) {
    i = i + 1;
    a[i] = 0;
}
```
we are looking for loop invariants that will show that something “bad” will not happen

```java
//@ loop_invariant a != null;
//@ loop_invariant i + 1 >= 0;

for (int i = 0; i < a.length - 1; i++) {
    i = i + 1;
    a[i] = 0;
}
```
//@ requires a != null;
/*@ requires
@ (\forall int x; (0 <= x & x < a.length) ==> a[x] != null); */
void setToZero(int [][] a) {
//@ loop_invariant
@ (\forall int x; (0 <= x & x < a.length) ==> a[x] != null);
//@ loop_invariant a != null;
//@ loop_invariant i >= 0; */
for (int i = 0; i < a.length; i++) {
//@ loop_invariant j >= 0;
@ loop_invariant a != null;
@ loop_invariant
@ (\forall int x; (0 <= x & x < a.length) ==> a[x] != null); */
for (int j = 0; j < a[i].length; j++)
    a[i][j] = 0;
}
}
let’s assume that desired behavior is expressed as assertions
using a weakest precondition calculus we can back-propagate
assertions to the head of the loop

```java
for (int i = 0; i < a.length - 1; i++) {
  i = i + 1;
  //@ assert a != null;
  //@ assert i <= 0;
  //@ assert i < a.length;
  a[i] = 0;
}
```
let’s assume that desired behavior is expressed as assertions

using a weakest precondition calculus we can back-propagate assertions to the head of the loop

```java
//@ loop_invariant (i < a.length - 1) ==> a != null;
for (int i = 0; i < a.length - 1; i++) {
    i = i + 1;
    //@ assert a != null;
    //@ assert i <= 0;
    //@ assert i < a.length;
    a[i] = 0;
}
```
let’s assume that desired behavior is expressed as assertions

using a weakest precondition calculus we can back-propagate assertions to the head of the loop

```java
//@ loop_invariant (i < a.length - 1) ==> a != null;
//@ loop_invariant (i < a.length - 1) ==> 0 <= i + 1;
```

```java
for (int i = 0; i < a.length - 1; i++) {
    i = i + 1;
    //@ assert a != null;
    //@ assert i <= 0;
    //@ assert i < a.length;
    a[i] = 0;
}
```
let's assume that desired behavior is expressed as assertions using a weakest precondition calculus we can back-propagate assertions to the head of the loop

```java
for (int i = 0; i < a.length - 1; i++) {
    i = i + 1;
    //@ assert a != null;
    //@ assert i <= 0;
    //@ assert i < a.length;
    a[i] = 0;
}
```
we take a *nest* of loops

\[
P_0; \\
\{ J_1 \} \textbf{ while } (C_1) \textbf{ do } \\
\ldots \\
P_{n-1}; \\
\{ J_n \} \textbf{ while } (C_n) \textbf{ do } \\
P_n; \\
\textbf{ assert } e;
\]

and propagate the invariant outwards

\[
J_n \equiv wlp(P_n, e) \\
J_i \equiv wlp(P_i, J_{i+1})
\]
Are the Invariants OK?

- all loops must *preserve* the pertaining invariant

\[ \models J_i \Rightarrow \text{wlp}(\text{LoopBody}_i, J_i) \]

- the invariant of the outermost loop must be established by the preceding command

\[ \models \text{wlp}(P_0, J_1) \]
Implementation in ESC/Java2

JML-annotated Java code

Java parsing

AST

GC generation

Guarded Commands

loop desugaring

desugared GC

VC generation

VC

proving

bugs

invariant
generation
Guarded Commands:

\[
\text{cmd} := \ x \leftarrow \text{expr} \mid \text{assume } f \mid \text{assert } f \mid \\
\text{cmd} \land \text{cmd} \mid \text{cmd } ; \text{cmd} \mid \{ J \} \text{while expr do cmd}
\]

\( J, f \) are first-order logic formulas

- captures JML-annotated Java, examples:
  - \textit{assert} pre — a precondition of a called method,
  - desired behavior, such as \textit{assert } \( a \neq \) null
  - \textit{assume} post — a postcondition of a called method
Tweaking the Algorithm

▶ **invariant strengthening**, heuristically altering the invariant when found that it does not preserve the pertaining loop, e.g.,

\[ I[v \mapsto v'], \text{ } v \text{ is free in } I \text{ and } v' \text{ is a fresh variable} \]

▶ applying **formula simplifications** to the inferred invariants

▶ when computing weakest precondition, ignoring commands that do not seem *relevant* (relying on a heuristic)
Experiences and Future Work

- works splendidly on the examples I wrote, *nevertheless*,
- little of existing code is verifiable
- generated invariants are not too big, *nevertheless*,
- there is still opportunity to prune away trivial invariants
- reporting invariants to the user would serve as valuable feedback
- extending the analysis to take into account assertions *after* the loop would make the analysis considerably more useful