Automatic Non-termination Analysis

of Imperative Programs

Diploma Thesis by
Helga Velroyen
helga@velroyen.de

Key Symposium
June 14th, 2007

Department of
Computing Science
Prof. Hähnle
Chalmers University of
Technology, Sweden

Me and my Thesis

- Helga Velroyen, helga@velroyen.de
- student of Computer Science and Mathematics
- at Aachen Technical University, Germany
- currently at Chalmers for the thesis
- supervising professor at RWTH: Prof. Jürgen Giesl
- supervising professor in Chalmers: Prof. Reiner Hähnle
- advisor: Philipp Rümmer

Gaussian Sum

Example for a Non-terminating Program

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

Gaussian Sum

Example for a Non-terminating Program

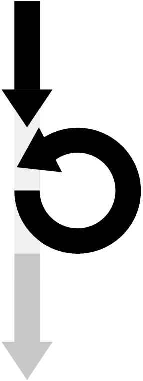
```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

Problem:

What happens if `i` has a negative value?

A common programming error.

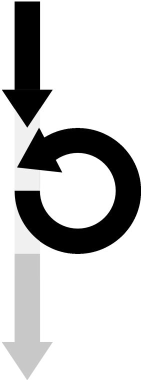
Invariants for Non-termination Detection



Find a boolean expression of program variables

1. that holds in the execution of the program right before the loop,
2. that implies the loop condition, and
3. that holds after an iteration of the loop body, in case it held before.

Invariants for Non-termination Detection



Find a boolean expression of program variables

1. that holds in the execution of the program right before the loop,
2. that implies the loop condition, and
3. that holds after an iteration of the loop body, in case it held before.

We call this expression invariant.
If we find an invariant, we have detected non-termination.

Formulae about Termination Behavior

(Non-)Termination expressed in Dynamic Logic

How to express termination in a formula:

$$\Rightarrow \langle p \rangle true$$

Formulae about Termination Behavior

(Non-)Termination expressed in Dynamic Logic

How to express termination in a formula:

$$\Rightarrow \langle p \rangle true$$

How to express non-termination in a formula:

$$\Rightarrow \neg\langle p \rangle true$$

$$\Rightarrow [p] false$$

Calculus Rule for Non-termination

How to treat non-termination in a proof

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U}\, inv, \Delta \\ inv, se \Rightarrow [p]\, inv \\ inv, \neg se \Rightarrow \varphi\end{array}}{\Gamma \Rightarrow \mathcal{U}\, [\ \texttt{while}\ (\ se\ )\ \{\ p\ \}\ ]\varphi, \Delta}\ \textsf{invRule}$$

Calculus Rule for Non-termination

How to treat non-termination in a proof

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U} inv, \Delta \\ inv, se \Rightarrow [p] inv \\ inv, \neg se \Rightarrow \varphi\end{array}}{\Gamma \Rightarrow \mathcal{U}\,[\,\texttt{while}\ (\ se\ )\ \{\ p\ \}\,]\varphi, \Delta}\ \textsf{invRule}$$

Calculus Rule for Non-termination

How to treat non-termination in a proof

$$\frac{\Gamma \Rightarrow \mathcal{U} inv, \Delta \qquad inv, se \Rightarrow [p] inv \qquad inv \Rightarrow se}{\Gamma \Rightarrow \mathcal{U}\,[\,\texttt{while}\ (\ se\ )\ \{\ p\ \}\,] false, \Delta}\ \textsf{invRule}$$

Calculus Rule for Non-termination

How to treat non-termination in a proof

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U}\mathit{inv}, \Delta \\ \mathit{inv}, \mathit{se} \Rightarrow [p]\mathit{inv} \\ \mathit{inv} \Rightarrow \mathit{se}\end{array}}{\Gamma \Rightarrow \mathcal{U}\,[\,\texttt{while (} \mathit{se} \texttt{) \{ } p \texttt{ \}}\,]\mathit{false}, \Delta}\ \mathsf{invRule}$$

Meaning of the three premisses:

- The invariant *inv* is initially valid.

Calculus Rule for Non-termination

How to treat non-termination in a proof

$$\frac{\Gamma \Rightarrow \mathcal{U}\,inv, \Delta \qquad inv, se \Rightarrow [p]\,inv \qquad inv \Rightarrow se}{\Gamma \Rightarrow \mathcal{U}\,[\,\texttt{while (} se \texttt{) \{ } p \texttt{ \} }\,]\mathit{false}, \Delta}\ \textsf{invRule}$$

Meaning of the three premisses:

- The invariant *inv* is initially valid.
- The invariant *inv* is preserved during body execution.

Calculus Rule for Non-termination

How to treat non-termination in a proof

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U}\, inv, \Delta \\ inv, se \Rightarrow [p]\, inv \\ \color{red}{inv \Rightarrow se}\end{array}}{\Gamma \Rightarrow \mathcal{U}\,[\, \texttt{while (}\ se\ \texttt{) \{}\ p\ \texttt{\}}\,]\mathit{false}, \Delta}\ \mathsf{invRule}$$

Meaning of the three premisses:

- The invariant *inv* is initially valid.
- The invariant *inv* is preserved during body execution.
- The invariant *inv* implies loop condition *se*.

Calculus Rule for Non-termination

How to treat non-termination in a proof

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U} inv, \Delta \\ inv, se \Rightarrow [p] inv \\ inv \Rightarrow se\end{array}}{\Gamma \Rightarrow \mathcal{U}\,[\,\texttt{while (}\ se\ \texttt{) \{}\ p\ \texttt{\}}\,] false, \Delta}\ \textsf{invRule}$$

Meaning of the three premisses:

- The invariant *inv* is initially valid.
- The invariant *inv* is preserved during body execution.
- The invariant *inv* implies loop condition *se*.

The invariant inv is not provided by the proof procedure.

Invariant Generator Tool

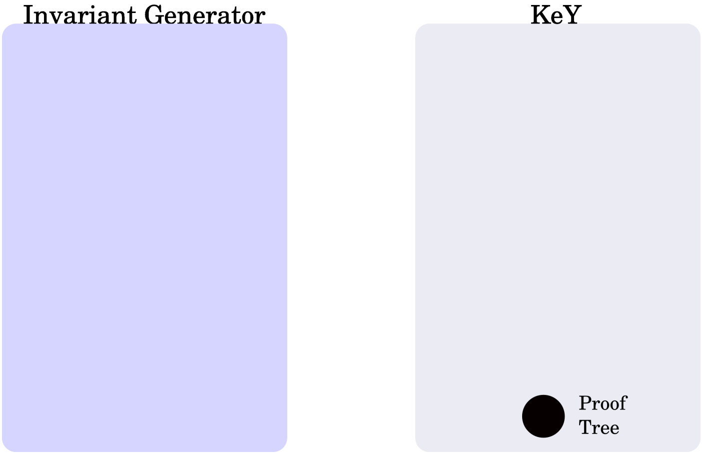
General Setting

Invariant Generator

KeY

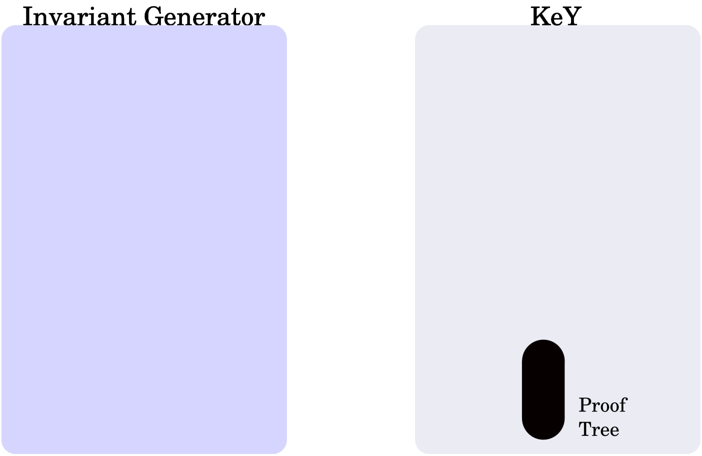
Invariant Generator Tool

General Setting



Invariant Generator Tool

General Setting

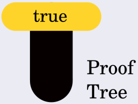


Invariant Generator Tool

General Setting

Invariant Generator

KeY

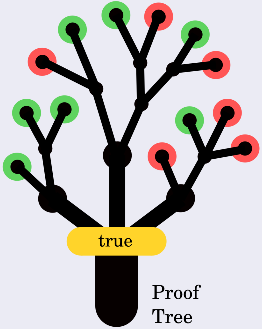


Invariant Generator Tool

General Setting

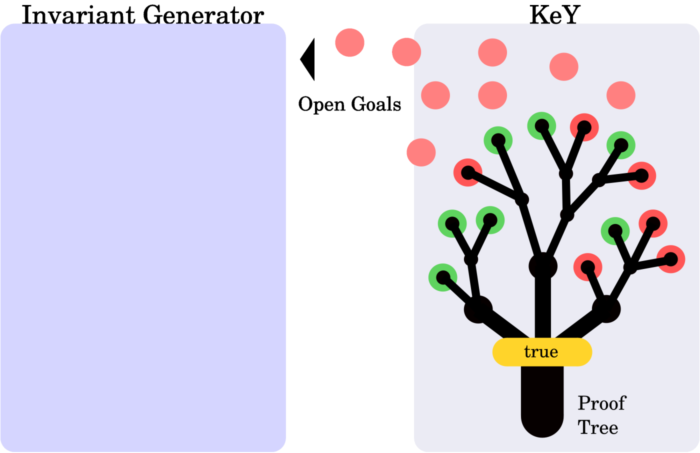
Invariant Generator

KeY



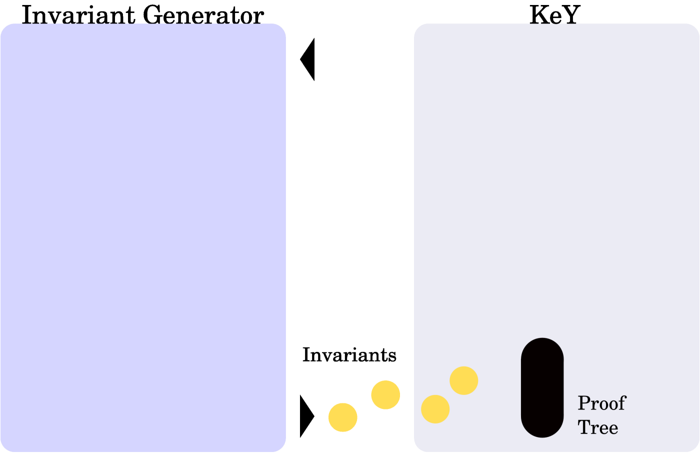
Invariant Generator Tool

General Setting



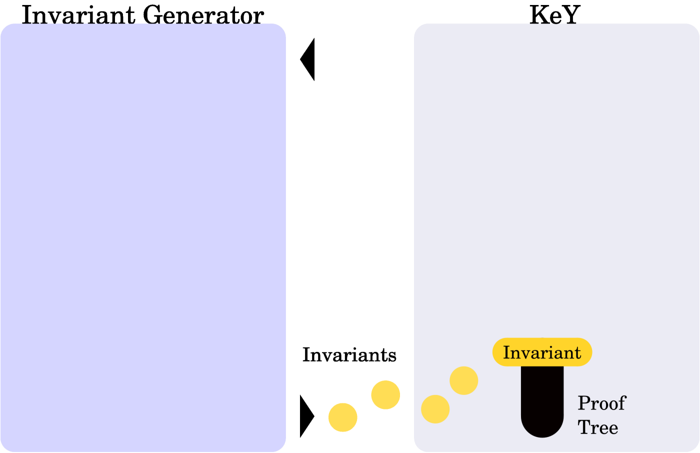
Invariant Generator Tool

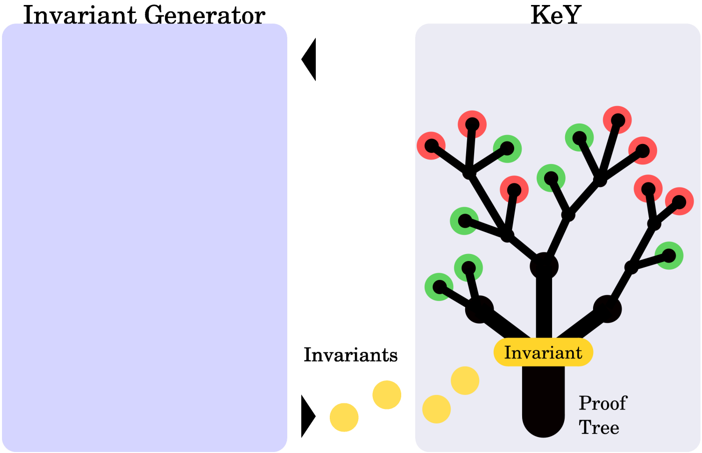
General Setting



Invariant Generator Tool

General Setting

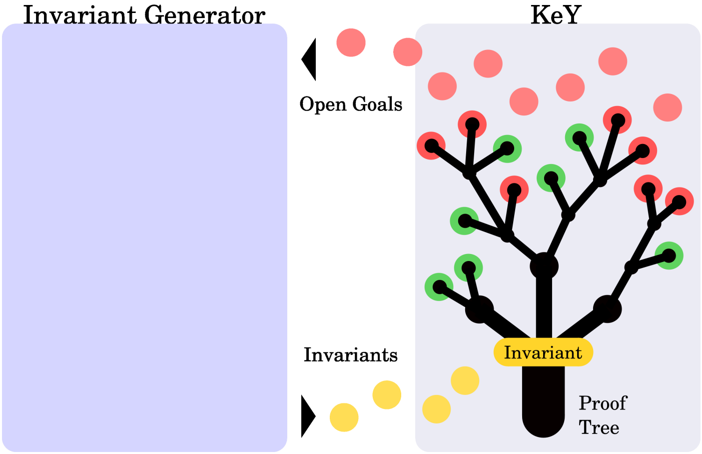




Invariant Generator
KeY
Invariants
Invariant
Proof
Tree

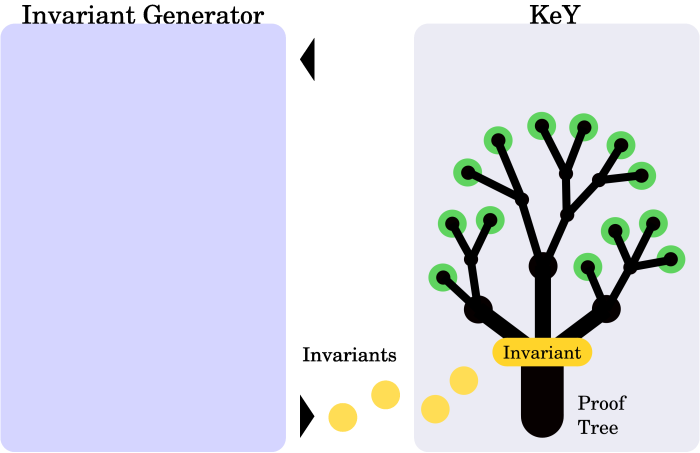
Invariant Generator Tool

General Setting



Invariant Generator Tool

General Setting



Soundness and Completeness

- The algorithm looks for non-termination only, not for termination.
- Soundness means: If the algorithm outputs:
  "The program does not terminate",
  then it actually does not terminate.
- The algorithm is sound, if the theorem prover is sound.

Soundness and Completeness

- The algorithm looks for non-termination only, not for termination.
- Soundness means: If the algorithm outputs:
  "The program does not terminate",
  then it actually does not terminate.
- The algorithm is sound, if the theorem prover is sound.
- If a program does not terminate, the output of the algorithm is either *"Does not terminate"* or *"I don't know if this program terminates"*.

Soundness and Completeness

- The algorithm looks for non-termination only, not for termination.
- Soundness means: If the algorithm outputs:
  "The program does not terminate",
  then it actually does not terminate.
- The algorithm is sound, if the theorem prover is sound.
- If a program does not terminate, the output of the algorithm is either *"Does not terminate"* or *"I don't know if this program terminates"*.
- If a program does terminate, the output of the algorithm is always *"I don't know if this program terminates"*

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

Proof obligation:

$$\exists l \; \{\texttt{i} := l\}[\texttt{Main.sum(i);}]\mathit{false}$$

Introduced meta variable:

$$\{\texttt{i} := I\}[\texttt{Main.sum(i);}]\mathit{false}$$

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*
- Open goal: $\mathtt{i} = 1 \Rightarrow$

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*
- Open goal: $\texttt{i} = 1 \Rightarrow$
- Invariant no 2: *true* && $\texttt{i} + 1$

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*
- Open goal: `i` $= 1 \Rightarrow$
- Invariant no 2: *true* && `i` $+ 1$
- Open goal: `i` $= 2 \Rightarrow$

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*
- Open goal: $\mathtt{i} = 1 \Rightarrow$
- Invariant no 2: *true* && $\mathtt{i} \neq 1$
- Open goal: $\mathtt{i} = 2 \Rightarrow$
- Invariant no 3: *true* && $\mathtt{i} > 1$

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*
- Open goal: $\texttt{i} = 1 \Rightarrow$
- Invariant no 2: *true* && $\texttt{i} \neq 1$
- Open goal: $\texttt{i} = 2 \Rightarrow$
- Invariant no 3: *true* && $\texttt{i} > 1$
- Open goal: $\texttt{i} = 2 \Rightarrow$

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*
- Open goal: $\texttt{i} = 1 \Rightarrow$
- Invariant no 2: *true* && $\texttt{i} \neq 1$
- Open goal: $\texttt{i} = 2 \Rightarrow$
- Invariant no 3: *true* && $\texttt{i} > 1$
- Open goal: $\texttt{i} = 2 \Rightarrow$
- Invariant no 3: *true* && $\texttt{i} < 1$

Invariant Creation

Example: Gaussian sum

```
int i = [...];

int sum = 0;

while (i != 0) {
  sum += i;
  i--;
}
```

- Invariant no 1: *true*
- Open goal: $\texttt{i} = 1 \Rightarrow$
- Invariant no 2: *true* && $\texttt{i} \neq 1$
- Open goal: $\texttt{i} = 2 \Rightarrow$
- Invariant no 3: *true* && $\texttt{i} > 1$
- Open goal: $\texttt{i} = 2 \Rightarrow$
- Invariant no 3: *true* && $\texttt{i} < 1$
- Proof closed with constraint: $I < 0$

Invariant Creation

Example: Up or Down

- Invariant no 1: *true*

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 1: *true*
- Open goal: $\mathtt{i} \leqslant 0 \Rightarrow$

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 1: *true*
- Open goal: $\texttt{i} \leqslant 0 \Rightarrow$
- Invariant no 2: *true* && $i > 0$

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 1: *true*
- Open goal: `i` $\leqslant 0 \Rightarrow$
- Invariant no 2: *true* && $i > 0$
- Open goal: `i` $= 1 \Rightarrow$

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 1: *true*
- Open goal: `i` $\leqslant 0 \Rightarrow$
- Invariant no 2: $true \ \&\& \ i > 0$
- Open goal: `i` $= 1 \Rightarrow$
- Invariant no 3: $true \ \&\& \ i > 0 \ \&\& \ i > 1$

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 1: *true*
- Open goal: `i` $\leqslant 0 \Rightarrow$
- Invariant no 2: $\mathit{true}\ \&\&\ i > 0$
- Open goal: `i` $= 1 \Rightarrow$
- Invariant no 3: $\mathit{true}\ \&\&\ i > 0\ \&\&\ i > 1$
- Open goal: `i` $= 2 \Rightarrow$

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 1: *true*
- Open goal: `i` $\leqslant 0 \Rightarrow$
- Invariant no 2: $true\ \&\&\ i > 0$
- Open goal: `i` $= 1 \Rightarrow$
- Invariant no 3: $true\ \&\&\ i > 0\ \&\&\ i > 1$
- Open goal: `i` $= 2 \Rightarrow$
- ...

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 1: *true*
- Open goal: `i` $\leqslant 0 \Rightarrow$
- Invariant no 2: true && $i > 0$
- Open goal: `i` $= 1 \Rightarrow$
- Invariant no 3: true && $i > 0$ && $i > 1$
- Open goal: `i` $= 2 \Rightarrow$
- ...
- Invariant no 7: true && $i > 0$ && $i > 1$ && ... && $i > 5$

Invariant Creation

Example: Up or Down

Smarter way: Introduce metavariables!

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 2: $\mathit{true} \;\&\&\; i > 0$
- Open goal: $\texttt{i} = 1 \Rightarrow$
- Invariant no 3: $\mathit{true} \;\&\&\; i > 0 \;\&\&\; i > 1$

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

Smarter way: Introduce metavariables!

- Invariant no 2: $true \; \&\& \; i > 0$
- Open goal: `i` $= 1 \Rightarrow$
- Invariant no 3: $true \; \&\& \; i > 0 \; \&\& \; i > 1$

Invariant Creation

Example: Up or Down

Smarter way: Introduce metavariables!

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 2: $\mathit{true}\ \&\&\ i > 0$
- Open goal: $\mathtt{i} = 1 \Rightarrow$
- Invariant no 3: $\mathit{true}\ \&\&\ i > 0\ \&\&\ i > M$

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 2: $\mathit{true}\ \&\&\ i > 0$
- Open goal: $\mathtt{i} = 1 \Rightarrow$
- Invariant no 3: $\mathit{true}\ \&\&\ i > 0\ \&\&\ i > M$
- Metavariables are treated as if they came from existentially quantified variables.

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 2: $true \;\&\&\; i > 0$
- Open goal: $\texttt{i} = 1 \Rightarrow$
- Invariant no 3: $true \;\&\&\; i > 0 \;\&\&\; i > M$

- Metavariables are treated as if they came from existentially quantified variables.
- "There is a lower bound M for i"

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

- Invariant no 2: $\mathit{true}\ \&\&\ i > 0$
- Open goal: $\mathtt{i} = 1 \Rightarrow$
- Invariant no 3:
  $\mathit{true}\ \&\&\ i > 0\ \&\&\ i > M$

- Metavariables are treated as if they came from existentially quantified variables.
- "There is a lower bound M for i"
- The constraint solver then tries to find it.

Invariant Creation

Example: Up or Down

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

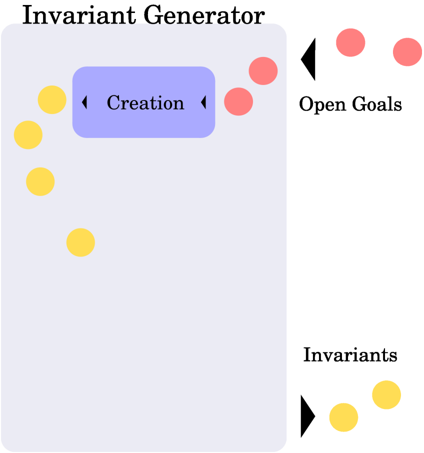
- Invariant no 2: $true \;\&\&\; i > 0$
- Open goal: $\mathtt{i} = 1 \Rightarrow$
- Invariant no 3: $true \;\&\&\; i > 0 \;\&\&\; i > M$

Proof closes with constraints:

$$M < I \;\&\&\; -1 < M \;\&\&\; 4 < M$$

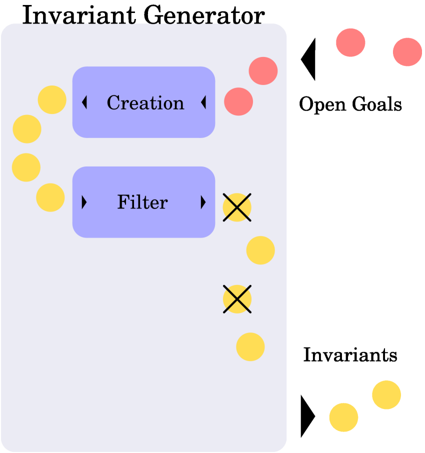
Invariant Generator Tool

Inner workings



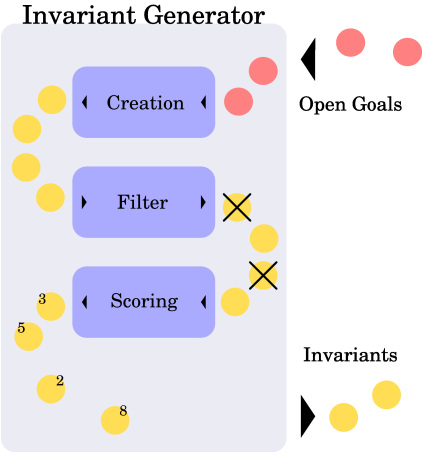
Invariant Generator Tool

Inner workings



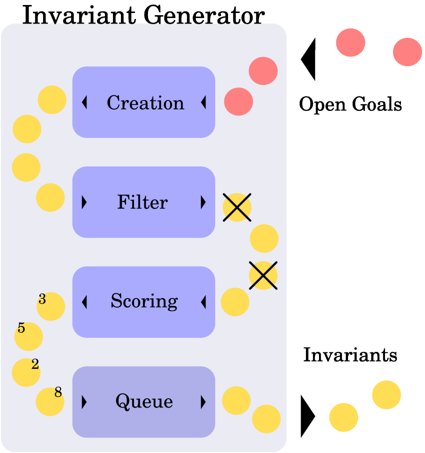
Invariant Generator Tool

Inner workings



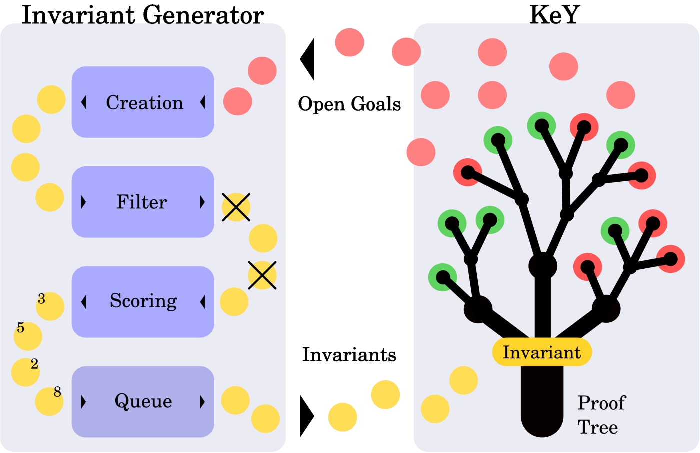
Invariant Generator Tool

Inner workings



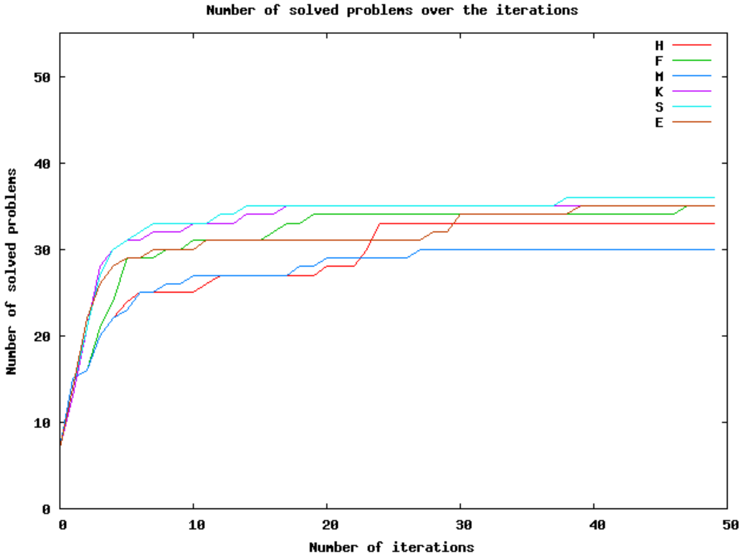
Invariant Generator Tool

Inner workings



Results

of the Experiments on WHILE Programs



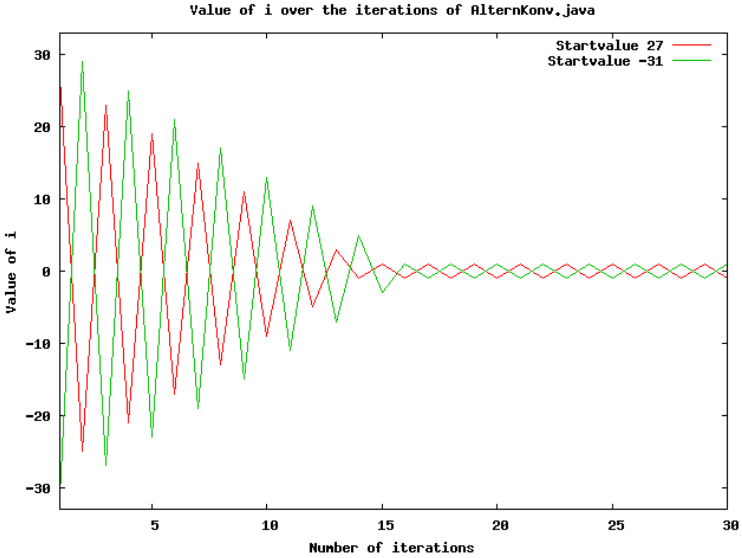
Example

Alternating and nearly konverging

```
while (i != 0) {
  if (i < 0) {
    i = i+2;
    if (i < 0) {
      i = i*(-1);
    }
  } else {
    i = i-2;
    if (i > 0) {
      i = i*(-1);
    }
  }
}
```

Example

Alternating and nearly konverging



Example

Alternating and nearly konverging

```
while (i != 0) {
  if (i < 0) {
    i = i+2;
    if (i < 0) {
      i = i*(-1);
    }
  } else {
    i = i-2;
    if (i > 0) {
      i = i*(-1);
    }
  }
}
```

Possible Invariants:

- $i \% 2 = 1$
- $i \% 2 = 1 \&\& i > -20$
- $i \% 2 = 1 \&\& i < 20$
- $i = 1 \,||\, i = -1$
- $\ldots$

Example

Alternating and nearly konverging

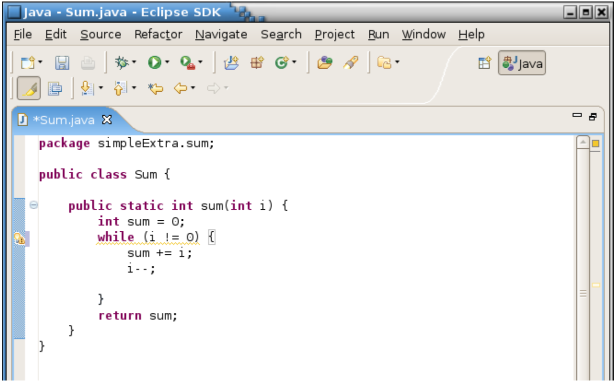
```
while (i != 0) {
  if (i < 0) {
    i = i+2;
    if (i < 0) {
      i = i*(-1);
    }
  } else {
    i = i-2;
    if (i > 0) {
      i = i*(-1);
    }
  }
}
```

Invariant found by tool:

$$\mathit{true}\ \&\&\ (i < 2)\ \&\&\ (i \neq 0)\ \&\&\ (i > -2)$$

Performance:

- between 7 and 28 iterations if solved
- some runs could not solve it


Java - Sum.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
*Sum.java
package simpleExtra.sum;
public class Sum {
public static int sum(int i) {
int sum = 0;
while (i != 0) {
sum += i;
i--;
}
return sum;
}
}

Thanks

Thank you for your attention.

Helga Velroyen
helga@velroyen.de