Towards Deductive Compilation: Implementing a Partial Evaluator Via a Software Verification Tool

Ran Ji
(joint work with Reiner Hähnle and Richard Bubel)

Chalmers University of Technology

1 April 2011

cost

Action IC0901

HATS
Highly Adaptable and Trustworthy Software using Formal Models

SEVENTH FRAMEWORK PROGRAMME

Introduction

Starting Point

Program verification tool (KeY) based on

- Dynamic logic for Java source code
- First-order theorem proving
- Symbolic execution
- Invariant reasoning

Introduction

Starting Point

Program verification tool (KeY) based on

- Dynamic logic for Java source code
- First-order theorem proving
- Symbolic execution
- Invariant reasoning

Constructing a specialized program from a verification proof attempt

Overview of Symbolic Execution

```
{a!=null && a.length>0}
h = a.length; ⇐ pc
l = 0;
while (a[(h-l)/2]>0) {
  body
}
rest
```

Overview of Symbolic Execution

```
{a!=null && a.length>0}
while (a[(h-1)/2]>0) {⇐
  body
}
rest
```

```
    ↓ a!=null && a.length>0
h=a.length
    ↓
    l=0  {h := a.length | l := 0}
```

1. Precondition is path condition in SE tree; nodes have symbolic state

Overview of Symbolic Execution

```
{a!=null && a.length>0}
int _i = a.length-0; ⇐
int _j = _i/2;
int _k = a[_j];
boolean _g = (_k>0);
while (_g) {
  body
}
rest
```

↓ a!=null && a.length>0

h=a.length

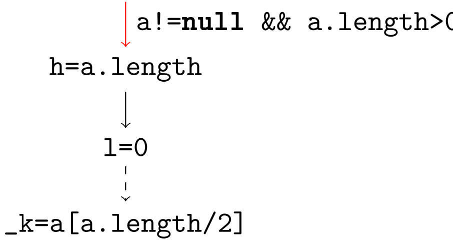
↓

l=0 {h := a.length | l := 0}

1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions

Overview of Symbolic Execution

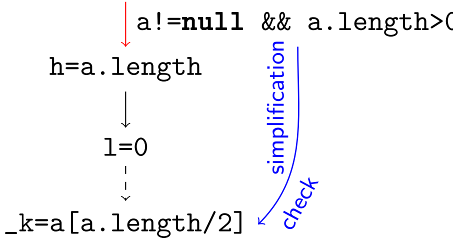
```
{a!=null && a.length>0}
boolean _g = (_k>0); ⇐
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions

Overview of Symbolic Execution

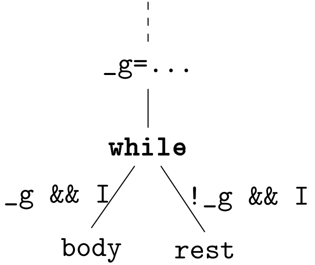
```
{a!=null && a.length>0}
boolean _g = (_k>0); ⇐
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions
3. First-order reasoning required for simplification, checking bounds

Overview of Symbolic Execution

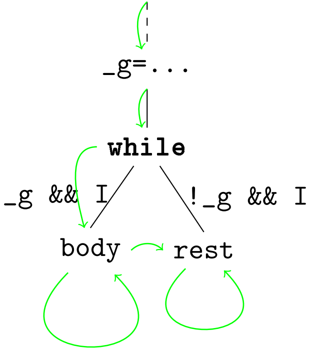
```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions
3. First-order reasoning required for simplification, checking bounds
4. Execute loop under suitable invariant

Overview of Symbolic Execution

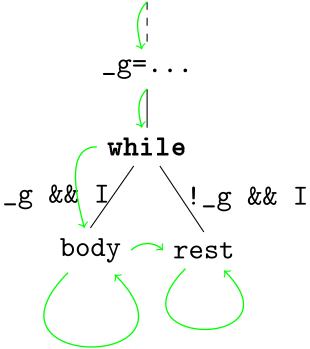
```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions
3. First-order reasoning required for simplification, checking bounds
4. Execute loop under suitable invariant
5. View SE as depth left first AST traversal

Overview of Symbolic Execution

```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```

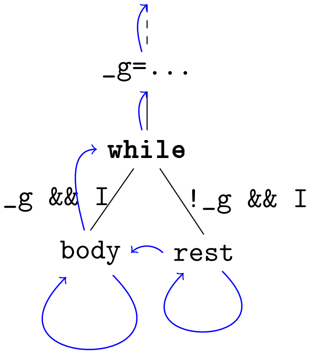


Observations

1. Transformation of complex assignments, symbolic state simplification: single static assignment (SSA) form easily obtainable
2. If strongest postcondition not needed, can use **true** as invariant

Overview of Symbolic Execution

```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```



Observations

1. Transformation of complex assignments, symbolic state simplification: single static assignment (SSA) form easily obtainable
2. If strongest postcondition not needed, can use **true** as invariant
3. May synthesize specialized program by bottom-up AST traversal:

Backward Analysis	used variables, etc.
Program Specialisation	dead code elimination, condition evaluation

Program Logic Calculus

Calculus

$$\text{ruleName} \quad \frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[p_1] \quad \dots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[p_n]}{\Gamma \Longrightarrow \mathcal{U}[p]}$$

Notation:

- Γ: path conditions (set of formulas)
- $\mathcal{U}$: update (information from the program has been excuted)
- p: Java program (program to be excuted)

Program Logic Calculus

Calculus

$$\text{ruleName} \quad \frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[p_1] \quad \dots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[p_n]}{\Gamma \Longrightarrow \mathcal{U}[p]}$$

Notation:

- Γ: path conditions (set of formulas)
- $\mathcal{U}$: update (information from the program has been excuted)
- p: Java program (program to be excuted)
- rule application from bottom-to-top
- postcondition ignored

Interleaving Symbolic Execution and Partial Evaluation

Proof-Search Space Reduction can be achieved by adding calculus rules performing (or invoking) a basic partial evaluator (FMCO 2009):

- constant propagation
- constant expression evaluation
- dead-code elimination

Interleaving Symbolic Execution and Partial Evaluation

Proof-Search Space Reduction can be achieved by adding calculus rules performing (or invoking) a basic partial evaluator (FMCO 2009):

- constant propagation
- constant expression evaluation
- dead-code elimination

One reason why this is a good idea:

Proof branching during symbolic execution creates new static input values:

$$\frac{\mathcal{U}(\mathtt{b}) \Longrightarrow \mathcal{U}[\mathtt{ack=true;r}] \quad \mathcal{U}(\neg\mathtt{b}) \Longrightarrow \mathcal{U}[\mathtt{ack=false;r}]}{\Longrightarrow \mathcal{U}[\mathtt{if\ (b)\ \{ack=true;\}\ else\ \{ack=false;\}\ r}]}$$

Interleaving Symbolic Execution and Partial Evaluation

Proof-Search Space Reduction can be achieved by adding calculus rules performing (or invoking) a basic partial evaluator (FMCO 2009):

- constant propagation
- constant expression evaluation
- dead-code elimination

One reason why this is a good idea:

Proof branching during symbolic execution creates new static input values:

$$\frac{\mathcal{U}(\mathtt{b}) \Longrightarrow \mathcal{U}[\mathtt{ack=true;r}] \quad \mathcal{U}(\neg\mathtt{b}) \Longrightarrow \mathcal{U}[\mathtt{ack=false;r}]}{\Longrightarrow \mathcal{U}[\mathtt{if\ (b)\ \{ack=true;\}\ else\ \{ack=false;\}\ r}]}$$

Can we extract a specialized program out of a verification proof?

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}]$$

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (\mathit{Fwd})(\mathit{Bk})$$

Sequent annotated with

$(\mathit{Fwd})(\mathit{Bk})$: program analysis and synthesis results from AST traversal

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (Fwd)(Bk)$$

Sequent annotated with

$(Fwd)(Bk)$: program analysis and synthesis results from AST traversal

- Fwd: program information maintained in forward analysis

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (\mathit{Fwd})(\mathit{Bk})$$

Sequent annotated with

$(\mathit{Fwd})(\mathit{Bk})$: program analysis and synthesis results from AST traversal

- *Fwd*: program information maintained in forward analysis
  - program variables potentially read in continuation of p

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (Fwd)(Bk)$$

Sequent annotated with

$(Fwd)(Bk)$: program analysis and synthesis results from AST traversal

- Fwd: program information maintained in forward analysis
  - program variables potentially read in continuation of $\mathtt{p}$
- $Bk = sp, use$: program information synthesized in backward analysis

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (Fwd)(Bk)$$

Sequent annotated with

$(Fwd)(Bk)$: program analysis and synthesis results from AST traversal

- Fwd: program information maintained in forward analysis
  - program variables potentially read in continuation of p
- $Bk = sp, use$: program information synthesized in backward analysis
  - sp: generated specialized program of Java source program p
  - use: program variables used in p (or continuation of p)

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (Fwd)(Bk)$$

Sequent annotated with

$(Fwd)(Bk)$: program analysis and synthesis results from AST traversal

- *Fwd*: program information maintained in forward analysis
  - program variables potentially read in continuation of p
- $Bk = sp, use$: program information synthesized in backward analysis
  - *sp*: generated specialized program of Java source program p
  - *use*: program variables used in p (or continuation of p)
- In general, *Fwd* and *Bk* could contain other information
  - View as specific pre-/postconditions or constraint system

Program Generation Rules

$$\frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[\mathrm{p}_1] \mid (X_1)(sp_1, use_1) \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[\mathrm{p}_n] \mid (X_n)(sp_n, use_n)}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (X)(sp, use)}$$

Program Generation Rules

$$\frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[\mathrm{p}_1] \mid (X_1)(sp_1, use_1) \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[\mathrm{p}_n] \mid (X_n)(sp_n, use_n)}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (X)(sp, use)}$$

symb.
exec.

- Java source code executed

Program Generation Rules

$$\frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[\mathrm{p}_1] \mid (X_1)(sp_1, use_1) \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[\mathrm{p}_n] \mid (X_n)(sp_n, use_n)}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (X)(sp, use)}$$

symb.
exec.

progr.
synth.

- Java source code executed then specialized program synthesized

Program Generation Rules

$$\frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[\mathrm{p}_1] \mid (X_1)(sp_1, use_1) \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[\mathrm{p}_n] \mid (X_n)(sp_n, use_n)}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (X)(sp, use)}$$

symb.
exec.

progr.
synth.

- Java source code executed then specialized program synthesized
- Establishing rule correctness requires to prove bisimulation property of original and specialized program

Selected Program Generation Rules

$$\text{emptyBox} \quad \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_, \_)}{\Gamma \Longrightarrow \mathcal{U}[\,] \mid (X)(\mathrm{nop}, X)}$$

- 'initiates' backward program synthesis

Selected Program Generation Rules

$$\text{emptyBox} \quad \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathrm{nop}, X)}$$

- 'initiates' backward program synthesis

Selected Program Generation Rules

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_, \_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

Selected Program Generation Rules

$$\text{emptyBox}\quad \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\texttt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\texttt{rest}] \mid (X)(\overline{rest}, use)}{\Gamma \Longrightarrow \mathcal{U}[\texttt{l = r;rest}] \mid (X)\begin{pmatrix} \texttt{l = r;}\overline{rest}, (use - \{l\}) \cup locs(r)) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{pmatrix}}$$

Selected Program Generation Rules

$$\text{emptyBox}\quad \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\mathtt{rest}] \mid (X)(\overline{rest}, use)}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{l\ =\ r;rest}] \mid (X)\begin{pmatrix} \mathtt{l = r;}\overline{rest}, (use - \{l\} \cup locs(r)) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{pmatrix}}$$

Selected Program Generation Rules

$$\text{emptyBox} \quad \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_, \_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\texttt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\texttt{rest}] \mid (X)(\overline{rest}, use)}{\Gamma \Longrightarrow \mathcal{U}[\texttt{l = r;rest}] \mid (X) \begin{pmatrix} \texttt{l} = \texttt{r}; \overline{rest}, (use - \{l\} \cup locs(r)) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{pmatrix}}$$

- updates used variable set
- assignment to unused variable deleted

Selected Program Generation Rules

emptyBox $\dfrac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\texttt{nop}, X)}$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\texttt{rest}] \mid (X)(\overline{\mathit{rest}}, \mathit{use})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{l = r;rest}] \mid (X) \begin{pmatrix} \texttt{l} = \texttt{r}; \overline{\mathit{rest}}, (\mathit{use} - \{l\} \cup \mathit{locs}(r)) & \text{if } l \in \mathit{use} \\ \overline{\mathit{rest}}, \mathit{use} & \text{otherwise} \end{pmatrix}}$$

- updates used variable set
- assignment to unused variable deleted

Conditional Rule

conditional

$$\frac{\begin{array}{l}\Gamma, \mathcal{U} b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ (\overline{p; rest}, use_{p;rest}) \\ \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ (\overline{q; rest}, use_{q;rest})\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}]}$$

$$\mid (X) \left(\begin{array}{l}\texttt{if (b) \{}\overline{\texttt{p; rest}}\texttt{\}} \\ \texttt{else \{}\overline{\texttt{q; rest}}\texttt{\}}\end{array}, (use_{p;rest} \cup use_{q;rest} \cup locs(b)) \right)$$

Conditional Rule

conditional

$$
\frac{\begin{array}{l}\Gamma, \mathcal{U} b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ \left(\overline{p;\ rest},\ use_{p;rest}\right) \\ \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ \left(\overline{q;\ rest},\ use_{q;rest}\right)\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}]}
$$

$$
\mid (X)\ \left(\begin{array}{l}\texttt{if (b) \{}\overline{\texttt{p; rest}}\texttt{\}} \\ \texttt{else \{}\overline{\texttt{q; rest}}\texttt{\}}\end{array},\ (use_{p;rest} \cup use_{q;rest} \cup locs(b))\right)
$$

Conditional Rule

conditional

$$\frac{\begin{array}{c}\Gamma, \mathcal{U} b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ (\overline{p;\mathit{rest}}, \mathit{use}_{p;\mathit{rest}}) \\ \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ (\overline{q;\mathit{rest}}, \mathit{use}_{q;\mathit{rest}})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}] \\ \qquad \mid (X) \left(\begin{array}{l}\texttt{if (b) } \{\overline{\texttt{p; rest}}\} \\ \texttt{else } \{\overline{\texttt{q; rest}}\}\end{array}, (\mathit{use}_{p;\mathit{rest}} \cup \mathit{use}_{q;\mathit{rest}} \cup \mathit{locs}(b)) \right)\end{array}}$$

Conditional Rule

conditional

$$\frac{\begin{array}{c}\Gamma, \mathcal{U} b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ (\overline{p;rest}, use_{p;rest}) \\ \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ (\overline{q;rest}, use_{q;rest})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}] \\ \qquad \mid (X) \left(\begin{array}{l}\texttt{if (b) } \{\overline{\texttt{p;rest}}\} \\ \texttt{else } \{\overline{\texttt{q;rest}}\}\end{array}, (use_{p;rest} \cup use_{q;rest} \cup locs(b))\right)\end{array}}$$

Generating Specialized Programs containing Loops

loopUnwind

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{if} (b) \{ p; \textbf{while} (b) p\} rest}] \quad \Downarrow(X)\ (\overline{\texttt{if(b)\{p;while(b)p\} rest}}, \mathit{use})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{while} (b) \{p\}; rest}] \quad \Downarrow(X)\ (\overline{\texttt{if(b)\{p;while(b)p\} rest}}, \mathit{use})}$$

Generating Specialized Programs containing Loops

loopUnwind

$$\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{if} (b) \{ p; \textbf{while} (b) p\} rest}] \mid (X)\ (\overline{\texttt{if(b)\{p; while(b)p\} rest}}\ , use)$$

$$\overline{\hphantom{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}$$

$$\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{while} (b) \{p\}; rest}] \mid (X)\ (\overline{\texttt{if(b)\{p; while(b)p\} rest}}\ , use)$$

Generating Specialized Programs containing Loops

loopUnwind

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{if} (b) \{ p; \textbf{while} (b) p\} rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{while} (b) \{p\}; rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use})}$$

Generating Specialized Programs containing Loops

loopUnwind

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{if} (b) \{ p; \textbf{while} (b) p\} rest}] \mid (X)\ \left(\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use}\right)}{\Gamma \Longrightarrow \mathcal{U}[\texttt{\textbf{while} (b) \{p\}; rest}] \mid (X)\ \left(\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use}\right)}$$

Loop Invariant Rule

$$\text{loopInvariant} \quad \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}\mathit{Inv} \\ \Gamma, \mathcal{U}\mathcal{V}_a(\mathit{Inv} \land b) \Longrightarrow [\mathtt{p}]\mathit{Inv} \\ \Gamma, \mathcal{U}\mathcal{V}_a(\mathit{Inv} \land \neg b) \Longrightarrow [\mathtt{rest}] \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Loop Invariant Rule

$$\text{loopInvariant}\quad \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\, \mathit{Inv} \\ \Gamma, \mathcal{U}\mathcal{V}_a(\mathit{Inv} \land b) \Longrightarrow [\mathtt{p}]\mathit{Inv} \\ \Gamma, \mathcal{U}\mathcal{V}_a(\mathit{Inv} \land \neg b) \Longrightarrow [\mathtt{rest}]\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Since we are not interested in proving correctness, use **`true`** as invariant!

Loop Invariant Rule

$$\text{loopInvariant}\quad \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\,Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\mathtt{p}]Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\mathtt{rest}]\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Since we are not interested in proving correctness, use **`true`** as invariant!

loopInvariantTrue

$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{p}, use_{body}) \\ \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \\ \qquad \mid (X)\ \left(\mathtt{while(b)\{\overline{p}\}\ \overline{rest}}, (use_{body} \cup use_{rest} \cup locs(b))\right)\end{array}}$$

Loop Invariant Rule

loopInvariant
$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\, Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\mathtt{p}]Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\mathtt{rest}]\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Since we are not interested in proving correctness, use **`true`** as invariant!

loopInvariantTrue
$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{\mathtt{p}}, use_{body}) \\ \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \\ \quad \mid (X)\ \left(\mathtt{while(b)\{\overline{p}\}}\ \overline{\mathtt{rest}}, (use_{body} \cup use_{rest} \cup locs(b))\right)\end{array}}$$

Loop Invariant Rule

$$\text{loopInvariant}\quad \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\, Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \land b) \Longrightarrow [\mathtt{p}] Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \land \neg b) \Longrightarrow [\mathtt{rest}]\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Since we are not interested in proving correctness, use **true** as invariant!

loopInvariantTrue

$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{p}, use_{body}) \\ \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \\ \qquad \mid (X)\ \left(\ \mathtt{while(b)\{\overline{p}\}}\ \overline{\mathtt{rest}}\ , (use_{body} \cup use_{rest} \cup locs(b))\right)\end{array}}$$

Loop Invariant Rule

$$\text{loopInvariant}\quad \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U} Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\mathtt{p}]Inv \\ \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\mathtt{rest}]\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

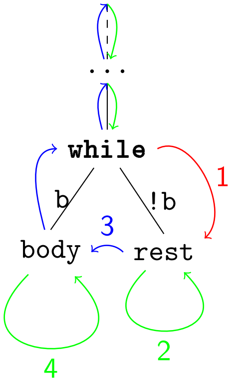
Since we are not interested in proving correctness, use **true** as invariant!

loopInvariantTrue

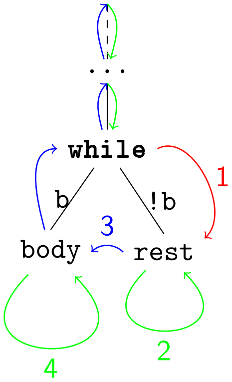
$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{p}, use_{body}) \\ \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \\ \qquad \mid (X)\ \left(\ \mathtt{while(b)\{\overline{p}\}}\ \overline{\mathtt{rest}}\ , (use_{body} \cup use_{rest} \cup locs(b))\right)\end{array}}$$

In "preserves invariant" branch the program variables used in the continuation of the loop body must be reflected correctly

Work Flow of Synthesizing Loop

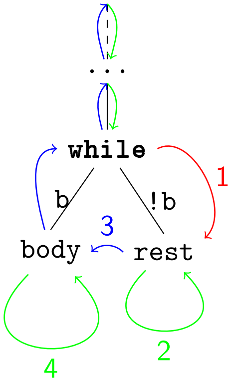


Work Flow of Synthesizing Loop



- Differs from traditional symbolic execution

Work Flow of Synthesizing Loop



- Differs from traditional symbolic execution
- Differs from strict forward/backward static analysis

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$\Longrightarrow$ [i=0;...] | $(tot)(sp_0, use_0)$

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$$\frac{\Longrightarrow \{\mathtt{i} := 0\}[\mathtt{count=n;\ldots}] \mid (tot)(sp_1, use_1)}{\Longrightarrow [\mathtt{i=0;\ldots}] \mid (tot)(sp_0, use_0)}$$

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$$\cfrac{\cfrac{\Longrightarrow \{\ldots \| \texttt{count} := \texttt{n}\}[\texttt{tot=0;while(i<=n)...}] \mid (tot)(sp_2, use_2)}{\Longrightarrow \{\texttt{i} := 0\}[\texttt{count=n;...}] \mid (tot)(sp_1, use_1)}}{\Longrightarrow [\texttt{i=0;...}] \mid (tot)(sp_0, use_0)}$$

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$$\dfrac{\dfrac{\dfrac{\Longrightarrow \{\ldots \| \texttt{tot} := 0\}[\texttt{while(i<=n)}\ldots] \mid (tot)(sp_3, use_3)}{\Longrightarrow \{\ldots \| \texttt{count} := \texttt{n}\}[\texttt{tot=0;while(i<=n)}\ldots] \mid (tot)(sp_2, use_2)}}{\Longrightarrow \{\texttt{i} := 0\}[\texttt{count=n;}\ldots] \mid (tot)(sp_1, use_1)}}{\Longrightarrow [\texttt{i=0;}\ldots] \mid (tot)(sp_0, use_0)}$$

Example Cont'd: Loop Unwind

$$\Longrightarrow \{\texttt{i} := 0 || \ldots || \texttt{tot} := 0\}[\texttt{while(i<=n)...}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\frac{\Longrightarrow \{\mathtt{i} := 0 || \ldots\}[\mathtt{if(i<=n)\{...;if(i>=2\ \&\&\ cpn)...;i++;while...\}}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)}{\Longrightarrow \{\mathtt{i} := 0 || \ldots || \mathtt{tot} := 0\}[\mathtt{while(i<=n)...}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots\}[\mathtt{if(0<=n)\{...;if(0>=2\ \&\&\ cpn)...;i=0+1;while...\}}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots\}[\mathtt{if(i<=n)\{...;if(i>=2\ \&\&\ cpn)...;i++;while...\}}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots || \mathtt{tot} := 0\}[\mathtt{while(i<=n)...}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots ; \mathtt{tot} := 0\}[\mathtt{if(0<=n)\{...;tot=0+m;i=1;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots\}[\mathtt{if(0<=n)\{...;if(0>=2\ \&\&\ cpn)...;i=0+1;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots\}[\mathtt{if(i<=n)\{...;if(i>=2\ \&\&\ cpn)...;i++;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots || \mathtt{tot} := 0\}[\mathtt{while(i<=n)...}] \mid (tot)(sp_3, use_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots || \mathtt{tot} := 0\}[\mathtt{if(0<=n)\{int\ m=read();tot=m;i=1;while...\}}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots ; \mathtt{tot} := 0\}[\mathtt{if(0<=n)\{...;tot=0+m;i=1;while...\}}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots\}[\mathtt{if(0<=n)\{...;if(0>=2\ \&\&\ cpn)...;i=0+1;while...\}}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots\}[\mathtt{if(i<=n)\{...;if(i>=2\ \&\&\ cpn)...;i++;while...\}}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

$$\Longrightarrow \{\mathtt{i} := 0 || \ldots || \mathtt{tot} := 0\}[\mathtt{while(i<=n)...}] \mid (\mathit{tot})(\mathit{sp}_3, \mathit{use}_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$\Longrightarrow \{\ldots\}[$`if(0<=n){int m=read();tot=m;i=1;while...}`$] \mid (tot)(sp_3, use_3)$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$\frac{\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot) \qquad 0 <= n \Longrightarrow \{\ldots\}[\texttt{int m=read();...}] \mid (tot)(sp_4, use_4)}{\Longrightarrow \{\ldots\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$\dfrac{\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot) \qquad \dfrac{\dfrac{0 <= n \Longrightarrow \{\ldots \| \texttt{m} := \texttt{read()} \| \texttt{tot} := \texttt{m} \| \texttt{i} := 1\}[\texttt{while(i<=n)...}] \mid (tot)(sp_5, use_5)}{\ldots}}{0 <= n \Longrightarrow \{\ldots\}[\texttt{int m=read();...}] \mid (tot)(sp_4, use_4)}}{\Longrightarrow \{\ldots\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot  = tot + m * 9 / 10;
 else
  tot  = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$
\dfrac{
\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot)
\qquad
\dfrac{\dfrac{\dfrac{0 <= n \Longrightarrow \{\ldots\}[\texttt{if(i<=n)...;while...}] \mid (tot)(sp_5, use_5)}{0 <= n \Longrightarrow \{\ldots \| \texttt{m} := \texttt{read()} \| \texttt{tot} := \texttt{m} \| \texttt{i} := 1\}[\texttt{while(i<=n)...}] \mid (tot)(sp_5, use_5)}}{\ldots}}{0 <= n \Longrightarrow \{\ldots\}[\texttt{int m=read();...}] \mid (tot)(sp_4, use_4)}
}{
\Longrightarrow \{\ldots\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)
}
$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot  = tot + m * 9 / 10;
 else
  tot  = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$\dfrac{\dfrac{\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot) \qquad \dfrac{\dfrac{\dfrac{\dfrac{1 <= n \Longrightarrow \{\ldots \| \mathtt{i} := 2\}[\mathtt{while(i<=n)\ldots}] \mid (tot)(sp_6, use_6)}{\ldots}}{0 <= n \Longrightarrow \{\ldots\}[\mathtt{if(i<=n)\ldots;while\ldots}] \mid (tot)(sp_5, use_5)}}{0 <= n \Longrightarrow \{\ldots \| \mathtt{m} := \mathtt{read()} \| \mathtt{tot} := \mathtt{m} \| \mathtt{i} := 1\}[\mathtt{while(i<=n)\ldots}] \mid (tot)(sp_5, use_5)}}{\ldots}}{0 <= n \Longrightarrow \{\ldots\}[\mathtt{int\ m=read();\ldots}] \mid (tot)(sp_4, use_4)}}{\Longrightarrow \{\ldots\}[\mathtt{if(0<=n)\{int\ m=read();tot=m;i=1;while\ldots\}}] \mid (tot)(sp_3, use_3)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$1 <= n \Longrightarrow \{\ldots \| \mathtt{i} := 2\}[\mathtt{while(i<=n)\ldots}] \mid (tot)(sp_6, use_6)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\frac{\ldots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \qquad \ldots, i <= n \Longrightarrow [\texttt{int} \ \ldots] \mid (tot \cup i \cup tot)(sp_7, use_7)}{1 <= n \Longrightarrow \{\ldots \| \texttt{i} := 2\}[\texttt{while(i<=n)}\ldots] \mid (tot)(sp_6, use_6)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\dfrac{\ldots, \neg(i <= n) \Longrightarrow [\,] \mid (tot)(nop, tot) \qquad \dfrac{\ldots \Longrightarrow \{\mathtt{m := read()}\}[\mathtt{if(cpn)\ldots}] \mid (tot \cup i)(sp_8, use_8)}{\ldots, i <= n \Longrightarrow [\mathtt{int\ \ \ldots}] \mid (tot \cup i \cup tot)(sp_7, use_7)}}{1 <= n \Longrightarrow \{\ldots \| \mathtt{i := 2}\}[\mathtt{while(i<=n)\ldots}] \mid (tot)(sp_6, use_6)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$
\dfrac{\dfrac{\ldots, cpn \Longrightarrow \ldots \mid (tot \cup i)(sp_9, use_9) \quad \ldots, \neg cpn \Longrightarrow \{\ldots\}[\texttt{tot=tot+m;}\ldots] \mid (tot \cup i)(sp_{10}, use_{10})}{\dfrac{\ldots \Longrightarrow \{\texttt{m} := \texttt{read()}\}[\texttt{if(cpn)}\ldots] \mid (tot \cup i)(sp_8, use_8)}{\ldots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \quad \ldots, i <= n \Longrightarrow [\texttt{int}\ \ldots] \mid (tot \cup i \cup tot)(sp_7, use_7)}}}{1 <= n \Longrightarrow \{\ldots \| \texttt{i} := 2\}[\texttt{while(i<=n)}\ldots] \mid (tot)(sp_6, use_6)}
$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\ldots \Longrightarrow \{\ldots \| \mathtt{tot} := \mathtt{tot+m}\}[\mathtt{i++;}] \mid (\mathit{tot} \cup i)(\mathit{sp}_{11}, \mathit{use}_{11})$$

$$\ldots, \mathit{cpn} \Longrightarrow \ldots \mid (\mathit{tot} \cup i)(\mathit{sp}_9, \mathit{use}_9) \quad \ldots, \neg\mathit{cpn} \Longrightarrow \{\ldots\}[\mathtt{tot=tot+m;}\ldots] \mid (\mathit{tot} \cup i)(\mathit{sp}_{10}, \mathit{use}_{10})$$

$$\ldots \Longrightarrow \{\mathtt{m} := \mathtt{read()}\}[\mathtt{if(cpn)}\ldots] \mid (\mathit{tot} \cup i)(\mathit{sp}_8, \mathit{use}_8)$$

$$\ldots, \neg(i <= n) \Longrightarrow [] \mid (\mathit{tot})(\mathit{nop}, \mathit{tot}) \qquad \ldots, i <= n \Longrightarrow [\mathtt{int}\ \ldots] \mid (\mathit{tot} \cup i \cup \mathit{tot})(\mathit{sp}_7, \mathit{use}_7)$$

$$1 <= n \Longrightarrow \{\ldots \| \mathtt{i} := 2\}[\mathtt{while(i<=n)}\ldots] \mid (\mathit{tot})(\mathit{sp}_6, \mathit{use}_6)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\ldots, cpn \Longrightarrow \ldots \mid (tot \cup i)(sp_9, use_9) \quad \dfrac{\dfrac{\ldots \Longrightarrow \{\ldots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{\ldots \Longrightarrow \{\ldots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\ldots, \neg cpn \Longrightarrow \{\ldots\}[\texttt{tot=tot+m;}\ldots] \mid (tot \cup i)(sp_{10}, use_{10})}}{\ldots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\ldots] \mid (tot \cup i)(sp_8, use_8)}$$

$$\dfrac{\ldots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \qquad \ldots, i <= n \Longrightarrow [\texttt{int}\ \ldots] \mid (tot \cup i \cup tot)(sp_7, use_7)}{1 <= n \Longrightarrow \{\ldots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\ldots] \mid (tot)(sp_6, use_6)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dfrac{\dfrac{\dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{\dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}}{\dfrac{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)}{\dots, i <= n \Longrightarrow [\texttt{int} \ \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}} \quad \dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot)$$

$$\dfrac{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \qquad \dots, i <= n \Longrightarrow [\texttt{int} \ \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}{1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

Synthesis

- sp_{12} : nop $\qquad$ use_{12} : $\{tot, i\}$

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \qquad \dfrac{\dfrac{\dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{\dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}}{\dfrac{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)}{\dots, i <= n \Longrightarrow [\texttt{int} \ \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}} \qquad \dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot)}{1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

Synthesis

- $sp_{12} : nop$ $\qquad$ $use_{12} : \{tot, i\}$
- $sp_{10} : tot = tot + m; i++;$ $\qquad$ $use_{10} : \{tot, i\}$

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\ldots, cpn \Longrightarrow \ldots \mid (tot \cup i)(sp_9, use_9) \quad \dfrac{\dfrac{\ldots \Longrightarrow \{\ldots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{\ldots \Longrightarrow \{\ldots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\ldots, \neg cpn \Longrightarrow \{\ldots\}[\texttt{tot=tot+m;}\ldots] \mid (tot \cup i)(sp_{10}, use_{10})}}{\dfrac{\ldots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\ldots] \mid (tot \cup i)(sp_8, use_8)}{\ldots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \qquad \ldots, i <= n \Longrightarrow [\texttt{int} \ \ldots] \mid (tot \cup i \cup tot)(sp_7, use_7)}}$$

$$1 <= n \Longrightarrow \{\ldots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\ldots] \mid (tot)(sp_6, use_6)$$

Synthesis

- $sp_{12} : nop$ $\quad use_{12} : \{tot, i\}$
- $sp_{10} : tot = tot + m; i++;$ $\quad use_{10} : \{tot, i\}$
- $sp_8 : if(cpn)\{tot = tot + m * 9/10; i++;\}$
  $else\{tot = tot + m; i++;\}$ $\quad use_8 : \{tot, i, cpn\}$

Example Cont'd: Loop Invariant True

$$\dfrac{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \qquad \dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \qquad \dfrac{\dfrac{\dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{\dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}}{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)}}{\dots, i <= n \Longrightarrow [\texttt{int}\ \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}}{1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

Synthesis

- sp_{12} : nop — use_{12} : $\{tot, i\}$
- sp_{10} : $tot = tot + m; i++;$ — use_{10} : $\{tot, i\}$
- sp_8 : $if(cpn)\{tot = tot + m * 9/10; i++;\}$ $else\{tot = tot + m; i++;\}$ — use_8 : $\{tot, i, cpn\}$
- sp_6 : $while(i <= n)\{int\ m = read();$ $if(cpn)\{tot = tot + m * 9/10; i++;\}$ $else\{tot = tot + m; i++;\}\}$ — use_6 : $\{tot, i, cpn\}$

Specialized Program

Specialized Java Code

```
tot = 0;
if(0 <= n) {
  int m = read();
  tot = m;
  if(1 <= n) {
    int m = read();
    tot = tot + m;
    i = 2;
    while(i <= n) {
      int m = read();
      if (cpn) {
        tot = tot + m * 9 / 10;
        i++;
      } else {
        tot = tot + m;
        i++;
      }
    }
  }
}
return tot;
```

Specialized Program

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 9 / 10;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Specialized Java Code

```
tot = 0;
if(0 <= n) {
  int m = read();
  tot = m;
  if(1 <= n) {
    int m = read();
    tot = tot + m;
    i = 2;
    while(i <= n) {
      int m = read();
      if (cpn) {
        tot = tot + m * 9 / 10;
        i++;
      } else {
        tot = tot + m;
        i++;
      }
    }
  }
}
return tot;
```

Bytecode Compilation

$$\frac{\Gamma \Longrightarrow \{l := r\}[\texttt{rest}]\ (\overline{rest}, use)}{\Gamma \Longrightarrow [\texttt{l=r;rest}] \left(\begin{array}{ll} \texttt{iload r}; \texttt{istore l}; \overline{rest}, ((use - \{l\}) \cup \{r\}) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{array} \right)}$$

Realise a rule-based Java bytecode compiler:

- Change the target language from Java source code to Java bytecode
- Single static assignment form: easy to synthesize bytecode
- Compiler correctness: soundness of program logic + local bisimulation
- Some available optimizations (FO reasoning, partial evaluation):
  - dead code elimination (can't reach unexecuted code in closed branches)
  - type inference
  - safety analysis (avoid creation of exception handlers)
  - constant propagation, expression simplification
  - precise usage, binding time analysis of variables

Summary

- New architecture for verified compilation:
  Verification + PE + local transformation = (verified) compilation
- Correctness of symbolic execution rules & bisimulation property guarantee correct specialiszation/compilation
- Symbolic execution permits dynamic analysis at compile time
- First-order reasoning, partial evaluation integrated
  - Infeasible path detection + interleaving partial evaluation:
    $\Rightarrow$ specialized and optimized programs
- Use-definition chains are maintained to eliminate unused assignments
  - Further analyses can be added
- Contracts (variable import/export) computed automatically:
  compositional (can compile methods independently)
- Implementation in KeY verification system ongoing

Related Work, Outlook

Related Work

- Compiler verification
- Hoare's Grand Challenge "The Verifying Compiler"
  $\Rightarrow$ "The Compiling Verifier"
- Rule-based compilation
- Translation validation of optimizing compilers
- Online partial evaluation

Outlook

- Parallelize independent code
- Room for heuristics:
  - to unwind or not to unwind
  - merge tails (e.g., by computing product program)
- Import information from other tools, e.g., invariants
- Inlining of wrappers/monitors on the fly