An Improved Rule for While Loops in Deductive Program Verification

Bernhard Beckert[1] Steffen Schlager[2] Peter H. Schmitt[2]

[1]Universität Koblenz-Landau

[2]Universität Karlsruhe

ICFEM 2005, Manchester

Outline

1. Preliminaries & Definitions
   - Program logic: Dynamic Logic for JAVA
   - Programs frames: Modifier Sets
   - State transitions: Updates
2. (Improved) Invariant Rule
3. An Invariant Rule for Total Correctness
4. An Invariant Rule for JavaCard

Program Logic – Dynamic Logic for JAVA

Syntax

- Basis: typed first-order logic
- Modal operators $[p]$ and $\langle p \rangle$ for each sequential JAVA program p

Program Logic – Dynamic Logic for JAVA

Syntax

- Basis: typed first-order logic
- Modal operators $[p]$ and $\langle p \rangle$ for each sequential JAVA program p

Semantics

- Semantics of p is a partial function
- Modal operators say something about the final state of p
- $[p]\,\phi$: If p terminates, then in its final state ϕ holds
  (partial correctness)
- $\langle p \rangle\,\phi$: p terminates and in its final state ϕ holds
  (total correctness)
- $\psi \rightarrow [p]\,\phi$ the same as Hoare triple $\{\psi\}\ p\ \{\phi\}$

Signature

Signature

- Signature Σ contains rigid and non-rigid function symbols.
  - Rigid functions are e.g. $+, -, 0, 1, \ldots$
  - Non-rigid functions are used to model program variables and arrays that are modified by programs, e.g. program variables, arrays, etc.
- A location is a non-rigid ground term that can be modified by a program, e.g. $a[0] = 5$;

Modifier Sets

- Specify locations that might be changed by a program

Definition (Modifier Set)

Let

- f^j a non-rigid function symbol, and
- $t^j_1, \ldots, t^j_{n_j}$ terms ($j \geq 1$).

Then, the set

$$\{ \quad f^1(t^1_1 \ldots, t^1_{n_1}) \ , \ldots, \quad f^k(t^k_1 \ldots, t^k_{n_k}) \ \}$$

of pairs is a modifier set.

Modifier Sets

- Specify locations that might be changed by a program

Definition (Modifier Set)

Let

- g^j be a Dynamic Logic formula,
- f^j a non-rigid function symbol, and
- $t^j_1, \ldots, t^j_{n_j}$ terms ($j \geq 1$).

Then, the set

$$\{\ \langle g^1, f^1(t^1_1 \ldots, t^1_{n_1})\rangle, \ldots,\ \langle g^k, f^k(t^k_1 \ldots, t^k_{n_k})\rangle\ \}$$

of pairs is a modifier set.

Example

Example

```
i=0; j=0;

while (i<length(a)) {
  a[i]=0;
  i=i+1;
}
```

	Modifier sets for the loop
correct:	$\{\langle true, i\rangle, \langle true, j\rangle, \langle 0 \le x < length(a), a[x]\rangle\}$

Example

Example

```
i=0; j=0;

while (i<length(a)) {
  a[i]=0;
  i=i+1;
}
```

	Modifier sets for the loop
correct:	$\{\langle true, i\rangle, \langle true, j\rangle, \langle 0 \leq x < length(a), a[x]\rangle\}$
not correct:	$\{\langle 0 \leq x < length(a), a[x]\rangle\}$

State Updates

- Classical DL: state changes represented by substitutions

Example

$\langle \mathsf{i{=}0;}\rangle \phi \leftrightarrow \phi_{\mathsf{i}}^{\mathsf{0}}$

State Updates

- Classical DL: state changes represented by substitutions

Example

$\langle \mathsf{i}{=}0; \rangle \phi \leftrightarrow \phi_{\mathsf{i}}^{0}$

- Aliasing in object-oriented languages causes case distinctions

Example

$$\mathsf{a[i]} \doteq 0 \rightarrow \langle \mathsf{a[j]}{=}1; \rangle \mathsf{a[i]} \not\doteq \mathsf{a[j]} \rightsquigarrow \begin{cases} \text{Case 1:} & \mathsf{i} \doteq \mathsf{j} \\ \text{Case 2:} & \mathsf{i} \not\doteq \mathsf{j} \end{cases}$$

State Updates

- Classical DL: state changes represented by substitutions

Example

$\langle \mathsf{i}{=}0; \rangle \phi \leftrightarrow \phi_{\mathsf{i}}^{0}$

- Aliasing in object-oriented languages causes case distinctions

Example

$$\mathsf{a[i]} \doteq 0 \rightarrow \langle \mathsf{a[j]}{=}1; \rangle \mathsf{a[i]} \not\doteq \mathsf{a[j]} \rightsquigarrow \begin{cases} \text{Case 1:} & \mathsf{i} \doteq \mathsf{j} \\ \text{Case 2:} & \mathsf{i} \not\doteq \mathsf{j} \end{cases}$$

- Case distinction not always necessary

State Updates

- Classical DL: state changes represented by substitutions

Example

$\langle \mathsf{i=0;} \rangle \phi \leftrightarrow \phi_{\mathsf{i}}^{0}$

- Aliasing in object-oriented languages causes case distinctions

Example

$$\mathsf{a[i]} \doteq 0 \rightarrow \langle \mathsf{a[j]=1;} \rangle \mathsf{a[i]} \not\doteq \mathsf{a[j]} \rightsquigarrow \begin{cases} \text{Case 1:} & \mathsf{i} \doteq \mathsf{j} \\ \text{Case 2:} & \mathsf{i} \not\doteq \mathsf{j} \end{cases}$$

- Case distinction not always necessary
- Idea: collect updates and do not apply until program has disappeared

State Updates

- Classical DL: state changes represented by substitutions

Example

$\langle \mathsf{i}{=}0; \rangle \phi \leftrightarrow \phi_{\mathsf{i}}^{0}$

- Aliasing in object-oriented languages causes case distinctions

Example

$$\mathsf{a[i]} \doteq 0 \rightarrow \langle \mathsf{a[j]}{=}1; \rangle \mathsf{a[i]} \not\doteq \mathsf{a[j]} \rightsquigarrow \begin{cases} \text{Case 1:} & \mathsf{i} \doteq \mathsf{j} \\ \text{Case 2:} & \mathsf{i} \not\doteq \mathsf{j} \end{cases}$$

- Case distinction not always necessary
- Idea: collect updates and do not apply until program has disappeared
- Allows simplification before application, updates sometimes cancel out previous ones

State Updates

Definition (Syntax of Updates)

For all non-rigid ground terms l, and all terms v, if ϕ is a formula, then $\{l := v\}\phi$ is a formula as well. The expressions $\{l := v\}$ are called updates.

State Updates

Definition (Syntax of Updates)

For all non-rigid ground terms l, and all terms v, if ϕ is a formula, then $\{l := v\}\phi$ is a formula as well. The expressions $\{l := v\}$ are called updates.

Definition (Semantics of Updates)

$s \models \{l := v\}\phi$ iff $s' \models \phi$ where
s' coincides with s except for the interpretation of l, which in s' has the same value as v in s.

Quantified Updates

Definition (Syntax of Quantified Updates)

Let

- $\{f(t_1, \ldots, t_n) := v\}$ be an update and
- g a DL formula

Then $\{g,\ f(t_1, \ldots, t_n) := v\}\phi$ is a DL formula as well.
The expression $\{g,\ f(t_1, \ldots, t_n) := v\}$ is called quantified update.

Invariant Rule for DL

Sequent Calculus Loop Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \qquad \mathit{Inv} \wedge \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv} \wedge \neg\epsilon \vdash [\beta]\phi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (\epsilon)\ \{\alpha\}\beta]\phi,\ \Delta}$$

Invariant Rule for DL

Sequent Calculus Loop Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \qquad \mathit{Inv} \wedge \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv} \wedge \neg\epsilon \vdash [\beta]\phi}{\Gamma \vdash \mathcal{U}[\mathtt{while}\ (\epsilon)\ \{\alpha\}\beta]\phi,\ \Delta}$$

- *Inv* holds in the beginning

Invariant Rule for DL

Sequent Calculus Loop Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \qquad \mathit{Inv} \wedge \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv} \wedge \neg\epsilon \vdash [\beta]\phi}{\Gamma \vdash \mathcal{U}[\mathtt{while}\ (\epsilon)\ \{\alpha\}\beta]\phi,\ \Delta}$$

- *Inv* holds in the beginning
- *Inv* is in fact an invariant of the loop body

Invariant Rule for DL

Sequent Calculus Loop Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \qquad \mathit{Inv} \wedge \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv} \wedge \neg\epsilon \vdash [\beta]\phi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (\epsilon)\ \{\alpha\}\beta]\phi,\ \Delta}$$

- *Inv* holds in the beginning
- *Inv* is in fact an invariant of the loop body
- *Inv* implies the postcondition if loop terminates

Invariant Rule for DL

Sequent Calculus Loop Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \qquad \mathit{Inv} \wedge \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv} \wedge \neg\epsilon \vdash [\beta]\phi}{\Gamma \vdash \mathcal{U}[\mathtt{while}\ (\epsilon)\ \{\alpha\}\beta]\phi,\ \Delta}$$

- *Inv* holds in the beginning
- *Inv* is in fact an invariant of the loop body
- *Inv* implies the postcondition if loop terminates

- Context $\Gamma, \Delta, \mathcal{U}$ must be omitted in 2nd and 3rd premiss

Invariant Rule for DL

Sequent Calculus Loop Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \qquad \mathit{Inv} \land \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv} \land \neg\epsilon \vdash [\beta]\phi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (\epsilon)\ \{\alpha\}\beta]\phi,\ \Delta}$$

- *Inv* holds in the beginning
- *Inv* is in fact an invariant of the loop body
- *Inv* implies the postcondition if loop terminates

- Context $\Gamma, \Delta, \mathcal{U}$ must be omitted in 2nd and 3rd premiss
- Context contains (parts of) precondition of the operation and global system invariant

Invariant Rule for DL

Sequent Calculus Loop Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \qquad \mathit{Inv} \wedge \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv} \wedge \neg\epsilon \vdash [\beta]\phi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (\epsilon)\ \{\alpha\}\beta]\phi,\ \Delta}$$

- *Inv* holds in the beginning
- *Inv* is in fact an invariant of the loop body
- *Inv* implies the postcondition if loop terminates

- Context $\Gamma, \Delta, \mathcal{U}$ must be omitted in 2nd and 3rd premiss
- Context contains (parts of) precondition of the operation and global system invariant
- Required context information must be added to invariant *Inv*

Example

Example

Precondition: $\neg a \doteq null$

```
int i=0;
while (i<length(a)) {
  a[i]=0;
  i=i+1;
}
```

Postcondition: $\forall x : int.(0 \leq x \leq length(a) \rightarrow a[x] \doteq 0)$

Loop Invariant:

Example

Example

Precondition: $\neg a \doteq null$

```
int i=0;
while (i<length(a)) {
  a[i]=0;
  i=i+1;
}
```

Postcondition: $\forall x : int.(0 \leq x \leq length(a) \rightarrow a[x] \doteq 0)$

Loop Invariant: $i \leq length(a) \wedge \forall x : int.(0 \leq x < i \rightarrow a[x] \doteq 0)$

Example

Example

Precondition: $\neg a \doteq \mathit{null}$

```
int i=0;
while (i<length(a)) {
  a[i]=0;
  i=i+1;
}
```

Postcondition: $\forall x : \mathit{int}.(0 \leq x \leq \mathit{length}(a) \rightarrow a[x] \doteq 0)$

Loop Invariant: $i \leq \mathit{length}(a) \wedge \forall x : \mathit{int}.(0 \leq x < i \rightarrow a[x] \doteq 0)$
$\wedge \neg a \doteq \mathit{null}$

Example

Example

Precondition: $\neg a \doteq null \wedge \phi_{Inv}$

```
int i=0;
while (i<length(a)) {
  a[i]=0;
  i=i+1;
}
```

Postcondition: $\forall x : int.(0 \leq x \leq length(a) \rightarrow a[x] \doteq 0) \wedge \phi_{Inv}$

Loop Invariant: $i \leq length(a) \wedge \forall x : int.(0 \leq x < i \rightarrow a[x] \doteq 0)$
$\wedge \neg a \doteq null$

Example

Example

Precondition: $\neg a \doteq null \land \phi_{Inv}$

```
int i=0;
while (i<length(a)) {
  a[i]=0;
  i=i+1;
}
```

Postcondition: $\forall x : int.(0 \leq x \leq length(a) \rightarrow a[x] \doteq 0) \land \phi_{Inv}$

Loop Invariant: $i \leq length(a) \land \forall x : int.(0 \leq x < i \rightarrow a[x] \doteq 0)$
$\land \neg a \doteq null \land \phi'_{Inv}$

Improved Invariant Rule – Motivation

We would like to have a rule that allows keeping as much context as possible!
It is sound to keep parts of context that are not modified by the loop.

How keeping unmodified Context?

Simply deleting affected formulas not possible for object-oriented languages due to aliasing!

How keeping unmodified Context?

Simply deleting affected formulas not possible for object-oriented languages due to aliasing!

Example

$$\underbrace{a[i] \doteq 0 \land a[j] \doteq 0}_{\textit{context}},\ \underbrace{a[i] \geq 0}_{\textit{invariant}} \vdash [\ \underbrace{a[i]\text{++};}_{\textit{loop body}}\ ]\ \underbrace{a[i] \geq 0}_{\textit{invariant}}$$

How keeping unmodified Context?

Simply deleting affected formulas not possible for object-oriented languages due to aliasing!

Example

$$\underbrace{\cancel{a[i] \doteq 0} \land a[j] \doteq 0}_{context},\ \underbrace{a[i] \geq 0}_{invariant} \vdash [\ \underbrace{a[i]{+}{+};}_{loop\ body}\ ]\ \underbrace{a[i] \geq 0}_{invariant}$$

How keeping unmodified Context?

Simply deleting affected formulas not possible for object-oriented languages due to aliasing!

Example

$$\underbrace{\cancel{a[i] \doteq 0} \wedge a[j] \doteq 0}_{\textit{context}},\ \underbrace{a[i] \geq 0}_{\textit{invariant}} \vdash [\ \underbrace{a[i]{+}{+};}_{\textit{loop body}}\ ]\ \underbrace{a[i] \geq 0}_{\textit{invariant}}$$

Anonymous updates wipe out context information about locations that are modified

How keeping unmodified Context?

Simply deleting affected formulas not possible for object-oriented languages due to aliasing!

Example

$$\underbrace{\cancel{a[i] \doteq 0} \land a[j] \doteq 0}_{\textit{context}},\ \underbrace{a[i] \geq 0}_{\textit{invariant}} \vdash [\ \underbrace{a[i]\text{++};}_{\textit{loop body}}\ ]\ \underbrace{a[i] \geq 0}_{\textit{invariant}}$$

Anonymous updates wipe out context information about locations that are modified

Example

$$a[i] \doteq 0 \land a[j] \doteq 0, \{a[i] := c\}a[i] \geq 0 \vdash \{a[i] := c\}[a[i]\text{++};]a[i] \geq 0$$

Improved Invariant Rule

Definition (Improved Invariant Rule)

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \epsilon) \vdash \mathcal{U}\mathcal{V}[\alpha]\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \neg\epsilon) \vdash \mathcal{U}\mathcal{V}[\omega]\phi,\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (\epsilon)\ \{\alpha\}\omega]\phi,\ \Delta}$$

where $\mathcal{V}$ is an anonymous update w.r.t. to a correct modifier set for the loop body α.

Improved Invariant Rule

Definition (Improved Invariant Rule)

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \epsilon) \vdash \mathcal{U}\mathcal{V}[\alpha]\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \neg\epsilon) \vdash \mathcal{U}\mathcal{V}[\omega]\phi,\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}[\mathtt{while}\ (\epsilon)\ \{\alpha\}\omega]\phi,\ \Delta}$$

where $\mathcal{V}$ is an anonymous update w.r.t. to a correct modifier set for the loop body α.

Advantages of this rule

- Context can be kept as far as possible
- Modifier set optional
- Usually loops modify only few locations
- Separating aspects of which locations change (modifier set) and how they change (invariant)

A Version for Total Correctness

- Induction proofs guarantee total correctness

A Version for Total Correctness

- Induction proofs guarantee total correctness
- Invariant rule only considers partial correctness

A Version for Total Correctness

- Induction proofs guarantee total correctness
- Invariant rule only considers partial correctness
- Idea: Proof that some integer term v (called variant) decreases with each loop iteration

A Version for Total Correctness

- Induction proofs guarantee total correctness
- Invariant rule only considers partial correctness
- Idea: Proof that some integer term v (called variant) decreases with each loop iteration
- More precisely:

A Version for Total Correctness

- Induction proofs guarantee total correctness
- Invariant rule only considers partial correctness
- Idea: Proof that some integer term v (called variant) decreases with each loop iteration
- More precisely:
    - $v \geq 0$ in the beginning

A Version for Total Correctness

- Induction proofs guarantee total correctness
- Invariant rule only considers partial correctness
- Idea: Proof that some integer term v (called variant) decreases with each loop iteration
- More precisely:
  - $v \geq 0$ in the beginning
  - v strictly decreases with each execution of the loop body

A Version for Total Correctness

- Induction proofs guarantee total correctness
- Invariant rule only considers partial correctness
- Idea: Proof that some integer term v (called variant) decreases with each loop iteration
- More precisely:
  - $v \geq 0$ in the beginning
  - v strictly decreases with each execution of the loop body
  - If $v \geq 0$ then $v \geq 0$ after each execution of the loop body

A Version for Total Correctness

- Induction proofs guarantee total correctness
- Invariant rule only considers partial correctness
- Idea: Proof that some integer term v (called variant) decreases with each loop iteration
- More precisely:
  - $v \geq 0$ in the beginning
  - v strictly decreases with each execution of the loop body
  - If $v \geq 0$ then $v \geq 0$ after each execution of the loop body
- Termination follows from the well-foundedness of the natural numbers $\mathbb{N}$, i.e. there is no infinite descending chain $n_0 > n_1 > n_2 > \cdots$ because every non-empty subset has a minimal element (namely 0 in this particular case).

Improved Invariant Rule with Termination

Improved Invariant Rule with Termination

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}(\mathit{Inv} \land v \geq 0),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \epsilon \land v \geq 0) \vdash \mathcal{U}\mathcal{V}\{v' := v\}\langle\alpha\rangle(\mathit{Inv} \land v \geq 0 \land v < v'),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \neg\epsilon) \vdash \mathcal{U}\mathcal{V}\langle\omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}\langle\mathtt{while}\ (\epsilon)\ \{\alpha\}\omega\rangle\phi,\ \Delta}$$

Improved Invariant Rule with Termination

Improved Invariant Rule with Termination

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}(Inv \land v \geq 0),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(Inv \land \epsilon \land v \geq 0) \vdash \mathcal{U}\mathcal{V}\{v' := v\}\langle\alpha\rangle(Inv \land v \geq 0 \land v < v'),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(Inv \land \neg\epsilon) \vdash \mathcal{U}\mathcal{V}\langle\omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}\langle \texttt{while}\ (\epsilon)\ \{\alpha\}\omega\rangle\phi,\ \Delta}$$

- *Inv* holds in the beginning and v is non-negativ

Improved Invariant Rule with Termination

Improved Invariant Rule with Termination

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}(\mathit{Inv} \wedge v \geq 0),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \epsilon \wedge v \geq 0) \vdash \mathcal{U}\mathcal{V}\{v' := v\}\langle\alpha\rangle(\mathit{Inv} \wedge v \geq 0 \wedge v < v'),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \neg\epsilon) \vdash \mathcal{U}\mathcal{V}\langle\omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}\langle\texttt{while}\ (\epsilon)\ \{\alpha\}\omega\rangle\phi,\ \Delta}$$

- *Inv* holds in the beginning and *v* is non-negativ
- *Inv* is in fact an invariant of the loop body, *v* stricly decreases, and the property "*v* is non-negativ" is preserved

Improved Invariant Rule with Termination

Improved Invariant Rule with Termination

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}(\mathit{Inv} \land v \geq 0),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \epsilon \land v \geq 0) \vdash \mathcal{U}\mathcal{V}\{v' := v\}\langle\alpha\rangle(\mathit{Inv} \land v \geq 0 \land v < v'),\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \neg\epsilon) \vdash \mathcal{U}\mathcal{V}\langle\omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}\langle\texttt{while}\ (\epsilon)\ \{\alpha\}\omega\rangle\phi,\ \Delta}$$

- *Inv* holds in the beginning and *v* is non-negativ
- *Inv* is in fact an invariant of the loop body, *v* stricly decreases, and the property "*v* is non-negativ" is preserved
- *Inv* implies the postcondition if loop terminates

Problems with JavaCard

- JavaCard is a real programming language with features that make verification more difficult.
- Invariant rule not sound for loops causing abrupt termination.

Problems with JavaCard

- JavaCard is a real programming language with features that make verification more difficult.
- Invariant rule not sound for loops causing abrupt termination.

Example

$$\frac{}{i = 0 \vdash \mathcal{U}[\texttt{while (true) \{break;\}}]i = 1}$$

Problems with JavaCard

- JavaCard is a real programming language with features that make verification more difficult.
- Invariant rule not sound for loops causing abrupt termination.

Example

$$\frac{i = 0 \vdash \mathcal{U}\texttt{true}}{i = 0 \vdash \mathcal{U}[\texttt{while (true) \{break;\}}]i = 1}$$

Problems with JavaCard

- JavaCard is a real programming language with features that make verification more difficult.
- Invariant rule not sound for loops causing abrupt termination.

Example

$$\frac{\begin{array}{l} i = 0 \vdash \mathcal{U}\texttt{true} \\ i = 0,\ \mathcal{U}\mathcal{V}(\texttt{true} \land \texttt{true}) \vdash \mathcal{U}\mathcal{V}[\texttt{break;}]\texttt{true} \end{array}}{i = 0 \vdash \mathcal{U}[\texttt{while (true) \{break;\}}]i = 1}$$

Problems with JavaCard

- JavaCard is a real programming language with features that make verification more difficult.
- Invariant rule not sound for loops causing abrupt termination.

Example

$$\frac{\begin{array}{l} i = 0 \vdash \mathcal{U}\texttt{true} \\ i = 0,\ \mathcal{U}\mathcal{V}(\texttt{true} \land \texttt{true}) \vdash \mathcal{U}\mathcal{V}[\texttt{break;}]\texttt{true} \\ i = 0,\ \mathcal{U}\mathcal{V}(\texttt{true} \land \neg\texttt{true}) \vdash \mathcal{U}\mathcal{V}[]i = 1 \end{array}}{i = 0 \vdash \mathcal{U}[\texttt{while (true) }\{\texttt{break;}\}]i = 1}$$

Solution

Definition (Improved Invariant Rule for JavaCard)

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \epsilon) \vdash \mathcal{U}\mathcal{V}[\alpha]\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \epsilon) \vdash \mathcal{U}\mathcal{V}[\alpha]_{\mathit{continue}}\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \epsilon) \vdash \mathcal{U}\mathcal{V}\langle\alpha\rangle_{\mathit{abruptly},\mathit{not\_continue}}\mathit{true} \rightarrow \mathcal{U}\mathcal{V}[\pi\alpha\omega]\phi,\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \land \neg\epsilon) \vdash \mathcal{U}\mathcal{V}[\pi\omega]\phi,\ \Delta\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \mathtt{while}\ (\epsilon)\ \{\alpha\}\ \omega]\phi,\ \Delta}$$

Solution

Definition (Improved Invariant Rule for JavaCard)

$$\frac{\begin{array}{l} \Gamma \vdash \mathcal{U}\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \epsilon) \vdash \mathcal{U}\mathcal{V}[\alpha]\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \epsilon) \vdash \mathcal{U}\mathcal{V}[\alpha]_{\mathit{continue}}\mathit{Inv},\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \epsilon) \vdash \mathcal{U}\mathcal{V}\langle\alpha\rangle_{\mathit{abruptly},\mathit{not\_continue}}\mathit{true} \rightarrow \mathcal{U}\mathcal{V}[\pi\alpha\omega]\phi,\ \Delta \\ \Gamma,\ \mathcal{U}\mathcal{V}(\mathit{Inv} \wedge \neg\epsilon) \vdash \mathcal{U}\mathcal{V}[\pi\omega]\phi,\ \Delta \end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \mathtt{while}\ (\epsilon)\ \{\alpha\}\ \omega]\phi,\ \Delta}$$

In KeY we have no additional modalities $[\ ]_{\mathit{continue}}, \langle\ \rangle_{\mathit{abruptly},\mathit{not\_continue}}$, rather the loop body α is transformed (see example).