An Improved Rule for While Loops in Deductive Program Verification

Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt

June 9, 2005

Traditional Invariant Rule

$$\frac{}{\Gamma \vdash \quad [\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\varphi}$$

Traditional Invariant Rule

$$\frac{\Gamma \vdash \quad \mathit{Inv}}{\Gamma \vdash \quad [\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\varphi}$$

1. *Inv* (some DL formula) holds at the beginning

Traditional Invariant Rule

$$\frac{\Gamma \vdash \quad \mathit{Inv} \qquad \mathit{Inv},\ \epsilon \vdash [\alpha]\mathit{Inv}}{\Gamma \vdash \quad [\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\varphi}$$

1. *Inv* (some DL formula) holds at the beginning
2. *Inv* is indeed an invariant

Traditional Invariant Rule

$$\frac{\Gamma \vdash \quad \mathit{Inv} \qquad \mathit{Inv},\ \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv},\ \neg\epsilon \vdash \varphi}{\Gamma \vdash \quad [\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha\ \mathtt{od}]\varphi}$$

1. *Inv* (some DL formula) holds at the beginning
2. *Inv* is indeed an invariant
3. *Inv* entails postcondition

Traditional Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U}\,\mathit{Inv} \qquad \mathit{Inv},\ \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv},\ \neg\epsilon \vdash \varphi}{\Gamma \vdash \mathcal{U}[\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha\ \mathtt{od}]\varphi}$$

1. *Inv* (some DL formula) holds at the beginning
2. *Inv* is indeed an invariant
3. *Inv* entails postcondition
4. version with updates

Traditional Invariant Rule

$$\frac{\Gamma \vdash \mathcal{U} \mathit{Inv} \qquad \mathit{Inv}, \epsilon \vdash [\alpha]\mathit{Inv} \qquad \mathit{Inv}, \neg\epsilon \vdash \varphi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\varphi}$$

1. *Inv* (some DL formula) holds at the beginning
2. *Inv* is indeed an invariant
3. *Inv* entails postcondition
4. version with updates

Usually we also have a Δ there which we omit here. Can be negated and put into Γ.

Problems

Actual rule in KeY more involved due to

Problems

Actual rule in KeY more involved due to

- taclet language (local, non-destructive)

  ```
  find (==> [while(#e) #s] post) replacewith ...
  ```

Problems

Actual rule in KeY more involved due to

- taclet language (local, non-destructive)

```
find (==> [while(#e) #s] post) replacewith ...
```

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv} \quad \Gamma, \mathcal{U}\mathit{Inv}, \mathcal{U}\epsilon \vdash \mathcal{U}[\alpha]\mathit{Inv} \quad \Gamma, \mathcal{U}\mathit{Inv}, \mathcal{U}\neg\epsilon \vdash \mathcal{U}\varphi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\varphi}$$

Problems

Actual rule in KeY more involved due to

- taclet language (local, non-destructive)

  `find (==> [while(#e) #s] post) replacewith ...`

$$\frac{\Gamma \vdash \mathcal{U} Inv \quad \Gamma, \mathcal{U} Inv, \mathcal{U}\epsilon \vdash \mathcal{U}[\alpha] Inv \quad \Gamma, \mathcal{U} Inv, \mathcal{U}\neg\epsilon \vdash \mathcal{U}\varphi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\varphi}$$

- Java programming language (abrupt termination)
  - break, (continue)
  - exceptions
  - return

Problems with Taclets

- context $\Gamma, \mathcal{U}$ cannot be thrown away
- not sound to use context information in the 2nd and 3rd premiss

Problems with Taclets

- context $\Gamma, \mathcal{U}$ cannot be thrown away
- not sound to use context information in the 2nd and 3rd premiss

Example

$$\frac{}{x \doteq 0 \not\vdash [\texttt{while } x \leq 5 \texttt{ do } x = x + 1;\ \texttt{od}]x \doteq 0}$$

Problems with Taclets

- context $\Gamma, \mathcal{U}$ cannot be thrown away
- not sound to use context information in the 2nd and 3rd premiss

Example

$$\frac{x \doteq 0 \vdash \mathit{Inv} \qquad x \doteq 0,\ \mathit{Inv},\ x \leq 5 \vdash [x = x + 1]\mathit{Inv} \qquad x \doteq 0,\ \mathit{Inv},\ \neg x \leq 5 \vdash x \doteq 0}{x \doteq 0 \nvdash [\texttt{while } x \leq 5 \texttt{ do } x = x + 1;\ \texttt{od}]x \doteq 0}$$

With $\mathit{Inv} \equiv \mathit{true}$ all premisses are valid but the conlusion is not.

Solution to the Taclet Problem

Anonymous update $\mathcal{V}$ that assigns fixed, unknown values to all locations.

Solution to the Taclet Problem

Anonymous update $\mathcal{V}$ that assigns fixed, unknown values to all locations.

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U}\mathit{Inv} \\ \Gamma,\ \mathcal{U}\mathcal{V}\mathit{Inv},\ \mathcal{U}\mathcal{V}\epsilon \vdash \mathcal{U}\mathcal{V}[\alpha]\mathit{Inv} \\ \Gamma,\ \mathcal{U}\mathcal{V}\mathit{Inv},\ \mathcal{U}\mathcal{V}\neg\epsilon \vdash \mathcal{U}\mathcal{V}\varphi\end{array}}{\Gamma \vdash \mathcal{U}[\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha\ \mathtt{od}]\varphi}$$

Can be written as taclet!

Solution to the Taclet Problem—Example

Example

$$\frac{}{x \doteq 0 \nvdash [\texttt{while } x \leq 5 \texttt{ do } x = x + 1; \texttt{ od}]x \doteq 0}$$

Solution to the Taclet Problem—Example

Example

$$\frac{\begin{array}{l} x \doteq 0 \vdash Inv \\ x \doteq 0,\ \{x := c\}Inv,\ \{x := c\}x \leq 5 \vdash \{x := c\}[x = x + 1]Inv \\ x \doteq 0,\ \{x := c\}Inv,\ \{x := c\}\neg x \leq 5 \vdash \{x := c\}x \doteq 0 \end{array}}{x \doteq 0 \not\vdash [\texttt{while}\ x \leq 5\ \texttt{do}\ x = x + 1;\ \texttt{od}]x \doteq 0}$$

Solution to the Taclet Problem—Example

Example

$$x \doteq 0 \vdash Inv$$

$$x \doteq 0,\ \{x := c\}Inv,\ \{x := c\}x \leq 5 \vdash \{x := c\}[x = x + 1]Inv$$

$$x \doteq 0,\ \{x := c\}Inv,\ \neg c \leq 5 \vdash c \doteq 0$$

$$\overline{x \doteq 0 \not\vdash [\texttt{while}\ x \leq 5\ \texttt{do}\ x = x + 1;\ \texttt{od}]x \doteq 0}$$

Depending on *Inv* at least one of the three premisses does not hold!

Solution to the Taclet Problem—Example

Example

$$\frac{x \doteq 0 \vdash \mathit{Inv} \qquad x \doteq 0,\ \{x := c\}\mathit{Inv},\ \{x := c\}x \leq 5 \vdash \{x := c\}[x = x + 1]\mathit{Inv} \qquad x \doteq 0,\ \{x := c\}\mathit{Inv},\ \neg c \leq 5 \vdash c \doteq 0}{x \doteq 0 \not\vdash [\texttt{while}\ x \leq 5\ \texttt{do}\ x = x + 1;\ \texttt{od}]x \doteq 0}$$

Depending on *Inv* at least one of the three premisses does not hold!

In fact we do not enumerate all locations and assign unknown values to them. Rather, we really use a special update. The update simplifier knows how to handle this special update, i.e. everything to the left of the special update must not be used for update simplification. This, in facts, is similar to throwing away the context—but can be expressed as a taclet.

Problem of Abrupt Termination

- Traditional rule does not consider abrupt termination
- KeY calculus does not distinguish non-termination and abrupt termination

Problem of Abrupt Termination

- Traditional rule does not consider abrupt termination
- KeY calculus does not distinguish non-termination and abrupt termination

Example

$$\frac{}{\Gamma \vdash \mathcal{U}[\texttt{while (exp) \{...break;...\}}]\varphi}$$

Problem of Abrupt Termination

- Traditional rule does not consider abrupt termination
- KeY calculus does not distinguish non-termination and abrupt termination

Example

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv} \qquad \mathit{Inv},\ \texttt{exp} \vdash \overbrace{[\ldots\texttt{break;}\ldots]\mathit{Inv}}^{\equiv \mathit{true}} \qquad \mathit{Inv},\ \neg\texttt{exp} \vdash \varphi}{\Gamma \vdash \mathcal{U}[\texttt{while (exp) \{...break;...\}}]\varphi}$$

2nd premiss trivially valid in case of abrupt termination!

Solution to Abrupt Termination Problem

- Program transformation of the loop that allows us to distinguish abrupt and non-termination!

Solution to Abrupt Termination Problem

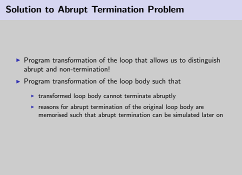
- Program transformation of the loop that allows us to distinguish abrupt and non-termination!
- Program transformation of the loop body such that

Solution to Abrupt Termination Problem

- Program transformation of the loop that allows us to distinguish abrupt and non-termination!
- Program transformation of the loop body such that
  - transformed loop body cannot terminate abruptly

Solution to Abrupt Termination Problem

- Program transformation of the loop that allows us to distinguish abrupt and non-termination!
- Program transformation of the loop body such that
  - transformed loop body cannot terminate abruptly
  - reasons for abrupt termination of the original loop body are memorised such that abrupt termination can be simulated later on



Instead of such a transformation one could also introduce new modalities to distinguish abstract and non-termination. But this has 2 major drawbacks:

- Less efficient since the calculus would have to execute the loop body twice: first within a normal box and second within the new modality
- lots of new calculus rule required for the additional modalities

An Example

```
while (i<100) {
   if (i==3)
     continue;
   j=j/i;
   i++;
}
```

An Example

```
while (i<100) {
  if (i==3)
    continue;
  j=j/i;
  i++;
}
```

⇝

```
boolean cont=false;
boolean exc=false;
java.lang.Throwable theExc;
try {
  body: {
    if (i<100) {
      if (i==3) {
          cont=true;
          break body;
      }
      j=j/i;
      i++;
    }
  }
} catch (java.lang.Throwable e) {
    exc=true;
    theExc=e;
}
```

Rule Respecting Abrupt Termination

Still simplified rule

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U} \mathit{Inv} \\ \mathit{Inv},\ \texttt{exp} \vdash [\alpha']((\neg \mathit{exc} \rightarrow \mathit{Inv}) \wedge (\mathit{exc} \rightarrow [\pi\ \texttt{throw theExc;}\ \omega]\varphi)) \\ \mathit{Inv},\ \neg\texttt{exp} \vdash [\pi\ \omega]\varphi\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{while(exp)}\ \{\ \alpha\ \}\ \omega]\varphi}$$

Rule Respecting Abrupt Termination

Still simplified rule

$$\frac{\Gamma \vdash \mathcal{U}\,\mathit{Inv} \qquad \mathit{Inv},\ \texttt{exp} \vdash [\alpha']((\neg \mathit{exc} \rightarrow \mathit{Inv}) \land (\mathit{exc} \rightarrow [\pi\ \texttt{throw theExc;}\ \omega]\varphi)) \qquad \mathit{Inv},\ \neg\texttt{exp} \vdash [\pi\ \omega]\varphi}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{while(exp)}\ \{\ \alpha\ \}\ \omega]\varphi}$$

We omit the anonymous updates and consider exceptions as the only source for abrupt terminations.

Improved Invariant Rule—Motivation

Example

```
int getMin(int [] a) {
  int m=a[0];
  int i=1;
  while (i<a.length) {
    if (a[i]<m)
      m=a[i];
    i++;
  }
  return m;
}
```

Improved Invariant Rule—Motivation

Example

```
int getMin(int[] a) {
  int m=a[0];
  int i=1;
  while (i<a.length) {
    if (a[i]<m)
      m=a[i];
    i++;
  }
  return m;
}
```

- postcondition:

  $$\varphi_{min} = (\forall x)(0 \leq x < a.length \rightarrow m \leq a[x])$$

Improved Invariant Rule—Motivation

Example

```
int getMin(int [] a) {
  int m=a[0];
  int i=1;
  while (i<a.length) {
    if (a[i]<m)
      m=a[i];
    i++;
  }
  return m;
}
```

- postcondition:

  $$\varphi_{min} = (\forall x)(0 \leq x < a.length \rightarrow m \leq a[x])$$

- additional part

  $$\varphi_{inv} = (\forall x)(0 \leq x < a.length \rightarrow a[x] = a'[x])$$

Improved Invariant Rule—Motivation

Example

```
int getMin(int [] a) {
  int m=a[0];
  int i=1;
  while (i<a.length) {
    if (a[i]<m)
      m=a[i];
    i++;
  }
  return m;
}
```

- postcondition:

  $$\varphi_{min} = (\forall x)(0 \leq x < a.length \rightarrow m \leq a[x])$$

- additional part

  $$\varphi_{inv} = (\forall x)(0 \leq x < a.length \rightarrow a[x] = a'[x])$$

- requires precodition φ_{inv}

  $$\varphi_{inv} \rightarrow [\mathrm{getMin(a)}](\varphi_{min} \wedge \varphi_{inv})$$

Improved Invariant Rule—Motivation

Example

```
int getMin(int [] a) {
  int m=a[0];
  int i=1;
  while (i<a.length) {
    if (a[i]<m)
      m=a[i];
    i++;
  }
  return m;
}
```

- obvious loop invariant

$$Inv = 0 \leq i \leq a.length \land (\forall x)(0 \leq x < i \rightarrow m \leq a[x])$$

Improved Invariant Rule—Motivation

Example

```
int getMin(int[] a) {
  int m=a[0];
  int i=1;
  while (i<a.length) {
    if (a[i]<m)
      m=a[i];
    i++;
  }
  return m;
}
```

- obvious loop invariant

$$Inv = 0 \leq i \leq a.length \land (\forall x)(0 \leq x < i \rightarrow m \leq a[x])$$

- *Inv* not strong enough

$$Inv, \neg i < a.length \not\vdash \varphi_{min} \land \varphi_{inv}$$

Improved Invariant Rule—Motivation

Example

```
int getMin(int [] a) {
  int m=a[0];
  int i=1;
  while (i<a.length) {
    if (a[i]<m)
      m=a[i];
    i++;
  }
  return m;
}
```

- not so obvious loop invariant

$$Inv = 0 \leq i \leq a.length \land (\forall x)(0 \leq x < i \rightarrow m \leq a[x]) \land \varphi_{inv}$$

- *Inv* not strong enough

$$Inv, \neg i < a.length \not\vdash \varphi_{min} \land \varphi_{inv}$$

Improved Invariant Rule—Motivation

A “right” invariant in general must express

- what the loop does

Improved Invariant Rule—Motivation

A “right” invariant in general must express

- what the loop does
- what the loop does *not* (change)

Improved Invariant Rule—Motivation

A “right” invariant in general must express

- what the loop does
- what the loop does *not* (change)

Improved Invariant Rule—Motivation

A “right” invariant in general must express

- what the loop does
- what the loop does *not* (change)
  Reason: Rule throws away context completely

Improved Invariant Rule—Motivation

A "right" invariant in general must express

- what the loop does
- what the loop does *not* (change)
  Reason: Rule throws away context completely

Ideas:

Improved Invariant Rule—Motivation

A "right" invariant in general must express

- what the loop does
- what the loop does *not* (change)
  Reason: Rule throws away context completely

Ideas:

- keeping context information about locations that are not changed within the loop is sound

Improved Invariant Rule—Motivation

A “right” invariant in general must express

- what the loop does
- what the loop does *not* (change)
  Reason: Rule throws away context completely

Ideas:

- keeping context information about locations that are not changed within the loop is sound
- use a more precise anonymous update that only wipes out locations that may change

Improved Invariant Rule—Motivation

A "right" invariant in general must express

- what the loop does
- what the loop does *not* (change)
  Reason: Rule throws away context completely

Ideas:

- keeping context information about locations that are not changed within the loop is sound
- use a more precise anonymous update that only wipes out locations that may change
- using modifier sets (assignable clauses in JML context) to precisely specify what the loop may change

Improved Invariant Rule—Motivation

A "right" invariant in general must express

- what the loop does
- what the loop does *not* (change)
  Reason: Rule throws away context completely

Ideas:

- keeping context information about locations that are not changed within the loop is sound
- use a more precise anonymous update that only wipes out locations that may change
- using modifier sets (assignable clauses in JML context) to precisely specify what the loop may change

A modifier set for methods has the following semantics:
After execution of the method, every location in the modifier set has the same value as in the beginning.
Analogously the same holds for loops.
In particular this means that the value of a location within the execution of the method, resp. loop body, can differ from the value in the beginning and end.

Improved Invariant Rule

Let $Mod = \{loc_1, loc_2, \ldots, loc_n\}$ be a modifier set for the loop, i.e. a set of locations that the loop possibly may change.

Improved Invariant Rule

Let $Mod = \{loc_1, loc_2, \ldots, loc_n\}$ be a modifier set for the loop, i.e. a set of locations that the loop possibly may change.

The update $\mathcal{V}_{Mod}$ is defined as

$$\mathcal{V}_{Mod} = \{loc_1 := c_1, loc_2 := c_2, \ldots, loc_n := c_2\}$$

where c_i are fresh constants.

Improved Invariant Rule

Let $Mod = \{loc_1, loc_2, \ldots, loc_n\}$ be a modifier set for the loop, i.e. a set of locations that the loop possibly may change.

The update $\mathcal{V}_{Mod}$ is defined as

$$\mathcal{V}_{Mod} = \{loc_1 := c_1, loc_2 := c_2, \ldots, loc_n := c_2\}$$

where c_i are fresh constants.

$$\frac{\begin{array}{l}\Gamma \vdash \mathcal{U} Inv \\ \Gamma,\ \mathcal{U}\mathcal{V}_{Mod} Inv,\ \mathcal{U}\mathcal{V}_{Mod}\epsilon \vdash \mathcal{U}\mathcal{V}_{Mod}[\alpha] Inv \\ \Gamma,\ \mathcal{U}\mathcal{V}_{Mod} Inv,\ \mathcal{U}\mathcal{V}_{Mod}\neg\epsilon \vdash \mathcal{U}\mathcal{V}_{Mod}\varphi\end{array}}{\Gamma \vdash \mathcal{U}[\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\ \texttt{od}]\varphi}$$

Improved Invariant Rule—Demo

Time for a Demo

Advantages: New Invariant Rule & Modifier Sets

- Seperate aspects of

Advantages: New Invariant Rule & Modifier Sets

- Seperate aspects of
  - which locations change (modifier set)

Advantages: New Invariant Rule & Modifier Sets

- Seperate aspects of
  - which locations change (modifier set)
  - how they change (loop invariant)

Advantages: New Invariant Rule & Modifier Sets

- Seperate aspects of
  - which locations change (modifier set)
  - how they change (loop invariant)
- (Optional) modifier set allow to state change information in a compact way

Advantages: New Invariant Rule & Modifier Sets

- Seperate aspects of
  - which locations change (modifier set)
  - how they change (loop invariant)
- (Optional) modifier set allow to state change information in a compact way
  - enumerate locations that may change instead of

Advantages: New Invariant Rule & Modifier Sets

- Seperate aspects of
  - which locations change (modifier set)
  - how they change (loop invariant)
- (Optional) modifier set allow to state change information in a compact way
  - enumerate locations that may change instead of
  - enumerate what does not change

Advantages: New Invariant Rule & Modifier Sets

- Seperate aspects of
  - which locations change (modifier set)
  - how they change (loop invariant)
- (Optional) modifier set allow to state change information in a compact way
  - enumerate locations that may change instead of
  - enumerate what does not change
- Make proof process more efficient

Conclusions

- improved invariant rule

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)
- implemented improved rule for JavaDL

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)
- implemented improved rule for JavaDL
- extended JML with an assignable clause for loops

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)
- implemented improved rule for JavaDL
- extended JML with an assignable clause for loops
- used quantified updates to talk about an unknown number of locations

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)
- implemented improved rule for JavaDL
- extended JML with an assignable clause for loops
- used quantified updates to talk about an unknown number of locations

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)
- implemented improved rule for JavaDL
- extended JML with an assignable clause for loops
- used quantified updates to talk about an unknown number of locations

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

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)
- implemented improved rule for JavaDL
- extended JML with an assignable clause for loops
- used quantified updates to talk about an unknown number of locations

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

solution (based on Philipp's proposal):

modifier set $Mod = \{0 \leq x < a.length \,?\, a[x],\ i\}$

Conclusions

- improved invariant rule
- proved soundness of improved rule (without abrupt termination)
- implemented improved rule for JavaDL
- extended JML with an assignable clause for loops
- used quantified updates to talk about an unknown number of locations

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

solution (based on Philipp's proposal):

modifier set $Mod = \{0 \leq x < a.length \, ? \, a[x], \; i\}$

anon. update $\mathcal{V}_{Mod} = \{0 \leq x < a.length \, ? \, a[x] := c_a[x], \; i := c_i\}$