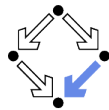


# Specifying and Verifying Programs (Part 2)

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

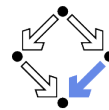
Research Institute for Symbolic Computation (RISC)  
Johannes Kepler University, Linz, Austria  
<http://www.risc.jku.at>



## 1. Programs as State Relations

## 2. The RISC ProgramExplorer

# Specification by State Predicates

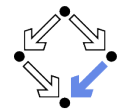


- Hoare calculus and predicate transformers use **state predicates**.
  - Formulas that talk about a single (pre/post-)state.
  - In such a formula, a reference  $x$  means “the value of program variable  $x$  in the given state”.
- Relationship** between pre/post-state is not directly expressible.
  - Requires uninterpreted mathematical constants.
 
$$\{x = a\}x := x + 1\{x = a + 1\}$$
- Unchanged variables** have to be explicitly specified.
 
$$\{x = a \wedge y = b\}x := x + 1\{x = a + 1 \wedge y = b\}$$
- The **semantics** of a command  $c$  is only **implicitly** specified.
  - Specifications depend on auxiliary state conditions  $P, Q$ .
 
$$\{P\}c\{Q\}$$

$$\text{wp}(c, Q) = P$$

Let us turn our focus from individual states to pairs of states.

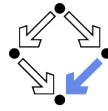
# Specification by State Relations



- We introduce formulas that denote **state relations**.
    - Talk about a pair of states (the pre-state and the post-state).
      - old  $x$ : “the value of program variable  $x$  in the pre-state”.
      - var  $x$ : “the value of program variable  $x$  in the post-state”.
  - We introduce the logical judgment  $c : [F]^{x, \dots}$ 
    - If the execution of  $c$  terminates normally, the resulting post-state is related to the pre-state as described by  $F$ .
    - Every variable  $y$  not listed in the set of variables  $x, \dots$  has the same value in the pre-state and in the post-state.
 
$$c : F \wedge \text{var } y = \text{old } y \wedge \dots$$
- $$x := x + 1 : [\text{var } x = \text{old } x + 1]^x$$
- $$x := x + 1 : \text{var } x = \text{old } x + 1 \wedge \text{var } y = \text{old } y \wedge \text{var } z = \text{old } z \wedge \dots$$

We will discuss the termination of commands later.

## State Relation Rules



$$\frac{c : [F]^{xs} \quad y \notin xs}{c : [F \wedge \text{var } y = \text{old } y]^{xs \cup \{y\}}}$$

**skip** :  $[\text{true}]^0$       **abort** :  $[\text{true}]^0$        $x = e : [\text{var } x = e']^{\{x\}}$

$$\frac{c_1 : [F_1]^{xs} \quad c_2 : [F_2]^{xs}}{c_1; c_2 : [\exists ys : F_1[ys/\text{var } xs] \wedge F_2[ys/\text{old } xs]]^{xs}}$$

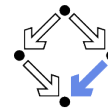
$$\frac{c : [F]^{xs}}{\text{if } e \text{ then } c : [\text{if } e' \text{ then } F \text{ else var } xs = \text{old } xs]^{xs}}$$

$$\frac{c_1 : [F_1]^{xs} \quad c_2 : [F_2]^{xs}}{\text{if } e \text{ then } c_1 \text{ else } c_2 : [\text{if } e' \text{ then } F_1 \text{ else } F_2]^{xs}}$$

$$\frac{c : [F]^{xs} \quad \vdash \forall xs, ys, zs : I[xs/\text{old } xs, ys/\text{var } xs] \wedge e[ys/xs] \wedge F[ys/\text{old } xs, zs/\text{var } xs] \Rightarrow I[xs/\text{old } xs, zs/\text{var } xs]}{\text{while } e \text{ do } \{l, t\} \quad c : [\neg e'' \wedge (I[\text{old } xs/\text{var } xs] \Rightarrow I)]^{xs}}$$

$\text{if } e \text{ then } F_1 \text{ else } F_2 \Leftrightarrow (e \Rightarrow F_1) \wedge (\neg e \Rightarrow F_2)$   
 $e' := e[\text{old } xs/xs], e'' := e[\text{var } xs/xs]$  (for all program variables  $xs$ )

## Example



$$c_1 = y := y + 1;$$

$$c_2 = x := x + y$$

$$c_1 : [\text{var } y = \text{old } y + 1]^y$$

$$c_2 : [\text{var } x = \text{old } x + \text{old } y]^x$$

$$c_1 : [\text{var } y = \text{old } y + 1 \wedge \text{var } x = \text{old } x]^{x,y}$$

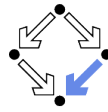
$$c_2 : [\text{var } x = \text{old } x + \text{old } y \wedge \text{var } y = \text{old } y]^{x,y}$$

$$c_1; c_2 : [\exists x_0, y_0 : y_0 = \text{old } y + 1 \wedge x_0 = \text{old } x \wedge \text{var } x = x_0 + y_0 \wedge \text{var } y = y_0]^{x,y}$$

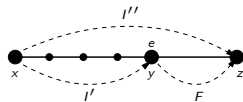
$$c_1; c_2 : [\text{var } x = \text{old } x + \text{old } y + 1 \wedge \text{var } y = \text{old } y + 1]^{x,y}$$

Mechanical translation and logical simplification.

## Loops



$$\frac{c : [F]^{xs} \quad \vdash \forall xs, ys, zs : I[xs/\text{old } xs, ys/\text{var } xs] \wedge e[ys/xs] \wedge F[ys/\text{old } xs, zs/\text{var } xs] \Rightarrow I[xs/\text{old } xs, zs/\text{var } xs]}{\text{while } e \text{ do } \{l, t\} \quad c : [\neg e'' \wedge (I[\text{old } xs/\text{var } xs] \Rightarrow I)]^{xs}}$$



$$w = \text{while } i < n \text{ do } \{l, t\} \quad (s := s + i; i := i + 1)$$

$$l \Leftrightarrow 0 \leq \text{var } i \leq \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{var } i-1} j$$

$$(s := s + i; i := i + 1) : [\text{var } s = \text{old } s + \text{old } i \wedge \text{var } i = \text{old } i + 1]^{s,i}$$

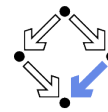
$$\vdash \forall s_x, s_y, s_z, i_x, i_y, i_z :$$

$$(0 \leq i_y \leq \text{old } n \wedge s_y = \sum_{j=0}^{i_y-1} j) \wedge i_y < \text{old } n \wedge (s_z = s_y + i_y \wedge i_z = i_y + 1) \Rightarrow 0 \leq i_z \leq \text{old } n \wedge s_z = \sum_{j=0}^{i_z-1} j$$

$$w : [(\text{var } i < \text{var } n) \wedge (0 \leq \text{old } i \leq \text{old } n \wedge \text{old } s = \sum_{j=0}^{\text{old } i-1} j \Rightarrow I)]^{s,i}$$

The loop relation is derived from the invariant (not the loop body); we have to prove the preservation of the loop invariant.

## Example



$$c =$$

$$\text{if } n < 0$$

$$s := -1$$

$$\text{else}$$

$$s := 0$$

$$i := 0$$

$$\text{while } i < n \text{ do } \{l, t\}$$

$$s := s + i$$

$$i := i + 1$$

$$l \Leftrightarrow 0 \leq \text{var } i \leq \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{var } i-1} j$$

$$t = \text{old } n - \text{old } i$$

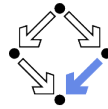
$$c : [\text{if old } n < 0$$

$$\text{then var } i = \text{old } i \wedge \text{var } s = -1$$

$$\text{else var } i = \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{old } n-1} j]^{s,i}$$

Let us calculate this "semantic essence" of the program.

## Example



```
c = if n < 0 then s := -1 else b
b = (s := 0; i := 0; w)
w = while i < n do {l, t} (s := s + i; i = i + 1)
```

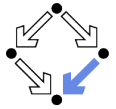
```
s := 0 : [var s = 0]s
s := 0 : [var s = 0 ∧ var i = old i]s,i
```

```
i := 0 : [var i = 0]i
i := 0 : [var i = 0 ∧ var s = old s]s,i
```

```
s := 0; i := 0 : [∃s0, i0 : s0 = 0 ∧ i0 = old i ∧ var i = 0 ∧ var s = s0]s,i
s := 0; i := 0 : [var s = 0 ∧ var i = 0]s,i
```

```
w : [¬(var i < var n) ∧ (0 ≤ old i ≤ old n ∧ old s = ∑j=0old i-1 j ⇒ l)]s,i
w : [var i ≥ old n ∧ (0 ≤ old i ≤ old n ∧ old s = ∑j=0old i-1 j ⇒ l)]s,i
```

## Example



```
c = if n < 0 then s := -1 else b
b = (s := 0; i := 0; w)
w = while i < n do {l, t} (s := s + i; i = i + 1)
```

```
s := 0; i := 0 : [var s = 0 ∧ var i = 0]s,i
w : [var i ≥ old n ∧ (0 ≤ old i ≤ old n ∧ old s = ∑j=0old i-1 j ⇒ l)]s,i
```

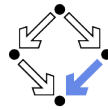
```
b : [∃s0, i0 : s0 = 0 ∧ i0 = 0 ∧
var i ≥ old n ∧ (0 ≤ i0 ≤ old n ∧ s0 = ∑j=0i0-1 j ⇒ l)]s,i
```

```
b : [∃s0, i0 : s0 = 0 ∧ i0 = 0 ∧
var i ≥ old n ∧ (0 ≤ old n ⇒ l)]s,i
```

```
b : [var i ≥ old n ∧
(0 ≤ old n ⇒ 0 ≤ var i ≤ old n ∧ var s = ∑j=0i-1 j)]s,i
```

```
b : [var i ≥ old n ∧
(0 ≤ old n ⇒ var i = old n ∧ var s = ∑j=0old n-1 j)]s,i
```

## Example



```
c = if n < 0 then s := -1 else b
b = (s := 0; i := 0; w)
w = while i < n do {l, t} (s := s + i; i = i + 1)
```

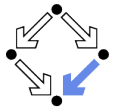
```
s := -1 : [var s = -1]s
s := -1 : [var i = old i ∧ var s = -1]s,i
```

```
b : [var i ≥ old n ∧
(0 ≤ old n ⇒ var i = old n ∧ var s = ∑j=0old n-1 j)]s,i
```

```
c : [if old n < 0
then var i = old i ∧ var s = -1
else var i ≥ old n ∧
(0 ≤ old n ⇒ var i = old n ∧ var s = ∑j=0old n-1 j)]s,i
```

```
c : [if old n < 0
then var i = old i ∧ var s = -1
else var i = old n ∧ var s = ∑j=0old n-1 j]s,i
```

## Partial Correctness



### ■ Specification (xs, P, Q)

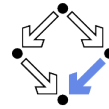
- Set of program variables xs (which may be modified).
- Precondition P (a formula with “old xs” but no “var xs”).
- Postcondition Q (a formula with both “old xs” and “var xs”).

### ■ Partial correctness of implementation c

1. Derive c : [F]<sup>xs</sup>.
2. Prove F ⇒ (P ⇒ Q)
  - Or: P ⇒ (F ⇒ Q)
  - Or: (P ∧ F) ⇒ Q

Verification of partial correctness leads to the proof of an implication.

## Relationship to Other Calculi



Let all state conditions refer via “old xs” to program variables xs.

### Hoare Calculus

- For proving  $\{P\}c\{Q\}$ ,
- it suffices to derive  $c : [F]^{xs}$
- and prove  $P \wedge F \Rightarrow Q[\text{var } xs/\text{old } xs]$ .

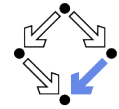
### Predicate Transformers

- Assume we can derive  $c : [F]^{xs}$ .
- If  $c$  does not contain loops, then
 
$$\text{wp}(c, Q) = \forall xs : F[xs/\text{var } xs] \Rightarrow Q[xs/\text{old } xs]$$

$$\text{sp}(c, P) = \exists xs : P[xs/\text{old } xs] \wedge F[xs/\text{old } xs, \text{old } xs/\text{var } xs]$$
- If  $c$  contains loops, the result is still a valid pre/post-condition but not necessarily the weakest/strongest one.

A generalization of the previously presented calculi.

## Termination



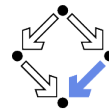
- We introduce a judgment  $c \downarrow T$ .

State condition  $T$  (a formula with “old xs” but no “var xs”).

- Starting with a pre-state that satisfies condition  $T$  the execution of command  $c$  terminates.
- Total correctness** of implementation  $c$ .  
Specification  $(xs, P, Q)$ .
  - Derive  $c \downarrow T$ .
  - Prove  $P \Rightarrow T$ .

Also verification of termination leads to the proof of an implication.

## Termination Condition Rules



**skip**  $\downarrow$  true      **abort**  $\downarrow$  true       $x := e \downarrow$  true

$$\frac{c_1 \downarrow T_1 \quad c_2 \downarrow T_2}{c_1; c_2 \downarrow T_1 \wedge \text{wp}(c_1, T_2)}$$

$$\frac{c \downarrow T}{\text{if } e \text{ then } c \downarrow e' \Rightarrow T}$$

$$\frac{c_1 \downarrow T_1 \quad c_2 \downarrow T_2}{\text{if } e \text{ then } c_1 \text{ else } c_2 \downarrow \text{if } e' \text{ then } T_1 \text{ else } T_2}$$

$$c : [F]^{xs} \quad c \downarrow T$$

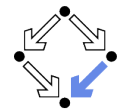
$\vdash \forall xs, ys, zs :$

$$I[xs/\text{old } xs, ys/\text{var } xs] \wedge e[ys/xs] \wedge F[ys/\text{old } xs, zs/\text{var } xs] \wedge t[ys/\text{old } xs] \geq 0 \Rightarrow T[ys/\text{old } xs] \wedge 0 \leq t[zs/\text{old } xs] < t[ys/\text{old } xs]$$

$$\text{while } e \text{ do } \{l, t\} \quad c \downarrow t \geq 0$$

In every iteration of a loop, the loop body must terminate and the termination term must decrease (but not become negative).

## Example



```

c =
  if n < 0
    s := -1
  else
    s := 0
    i := 0
    while i < n do {l,t}
      s := s + i
      i := i + 1
    
```

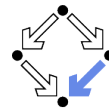
$$l \Leftrightarrow 0 \leq \text{var } i \leq \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{var } i-1} j$$

$$t = \text{old } n - \text{old } i$$

$c \downarrow$  if old  $n < 0$  then true else ...  
 $c \downarrow$  if old  $n < 0$  then true else old  $n \geq 0$   
 $c \downarrow$  true

We still have to prove the constraint on the loop iteration.

## Example



$s := s + i; i := i + 1 \downarrow \text{true}$

$\forall s_x, s_y, s_z, i_x, i_y, i_z :$

$(0 \leq i_y \leq \text{old } n \wedge s_y = \sum_{j=0}^{i_y-1} j) \wedge$

$i_y < \text{old } n \wedge$

$(s_z = s_y + i_y \wedge i_z = i_y + 1) \wedge$

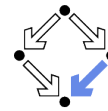
$\text{old } n - i_y \geq 0 \Rightarrow$

$\text{true} \wedge$

$0 \leq \text{old } n - i_z < \text{old } n - i_y$

Also this constraint is simple to prove.

## Abortion



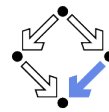
Also abortion can be ruled out by proving side conditions in the usual way.

Wolfgang Schreiner. *Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs*. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2011.

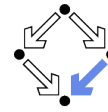
See the report for the full calculus.

## 1. Programs as State Relations

## 2. The RISC ProgramExplorer



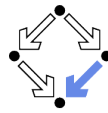
## The RISC ProgramExplorer



- An integrated environment for program reasoning.
  - Research Institute for Symbolic Computation (RISC), 2008–.  
<http://www.risc.jku.at/research/formal/software/ProgramExplorer>
  - Integrates the RISC ProofNavigator for computer-assisted proving.
  - Written in Java, runs under Linux (only), freely available (GPL).
- Programs written in “MiniJava”.
  - Subset of Java with full support of control flow interruptions.
  - Value (not pointer) semantics for arrays and objects.
- Theories and specifications written in a formula language.
  - Derived from the language of the RISC ProofNavigator.
- Semantic analysis and verification.
  - Program methods are translated into their “semantic essence”.
    - Open for human inspection.
  - From the semantics, the verification tasks are generated.
    - Solved by automatic decision procedure or interactive proof.

Tight integration of executable programs, declarative specifications, mathematical semantics, and verification tasks.

## Using the Software



See “The RISC ProgramExplorer: Tutorial and Manual”.

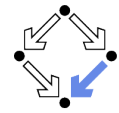
- **Develop a theory.**
  - File “*Theory.theory*” with a theory *Theory* of mathematical types, constants, functions, predicates, axioms, and theorems.
  - Can be also added to a program file.
- **Develop a program.**
  - File “*Class.java*” with a class *Class* that contains class (static) and object (non-static) variables, methods and constructors.
  - Class may be annotated by a theory (and an object invariant).
  - Methods may be annotated by method specifications.
  - Loops may be annotated by invariants and termination terms.
- **Analyze method semantics.**
  - Transition relations, termination conditions, ... of the method body and its individual commands.
- **Perform verification tasks.**
  - Frame, postcondition, termination, preconditions, loop-related tasks, type-checking conditions.

Wolfgang Schreiner

<http://www.risc.jku.at>

21/46

## Starting the Software



- **Starting the software:**  
module load ProgramExplorer (only at RISC)  
ProgramExplorer &
- **Command line options:**  
Usage: ProgramExplorer [OPTION]...  
OPTION: one of the following options:
  - h, --help: print this message.
  - cp, --classpath [PATH]:  
directories representing top package.

Environment Variables:

PE\_CLASSPATH:

the directories (separated by ".") representing the top package (default the current working directory)

...

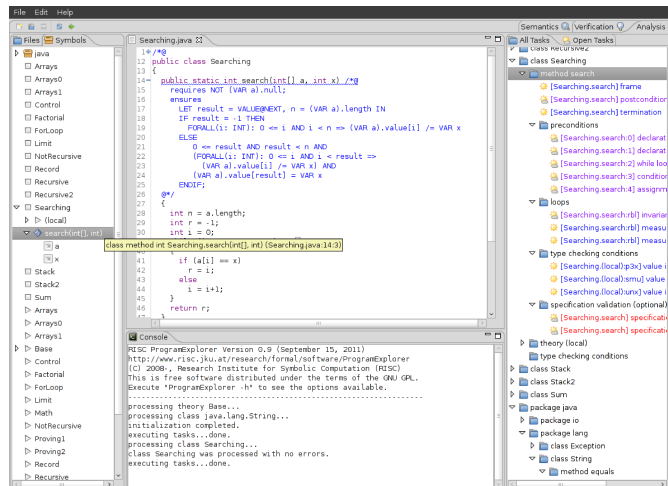
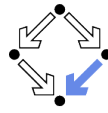
- **Task repository created/read in current working directory:**  
Subdirectory *.PETASKS.timestamp* (ProgramExplorer tasks)  
Subdirectory *.ProofNavigator* (ProofNavigator legacy)

Wolfgang Schreiner

<http://www.risc.jku.at>

22/46

## The Graphical User Interface

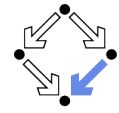


Wolfgang Schreiner

<http://www.risc.jku.at>

23/46

## A Program



```
/*@..  
class Sum  
{  
  static int sum(int n) /*@..  
  {  
    int s;  
    if (n < 0)  
      s = -1;  
    else  
    {  
      s = 0;  
      int i = 1;  
      while (i <= n) /*@..  
      {  
        s = s+i;  
        i = i+1;  
      }  
    }  
    return s;  
  }  
}
```

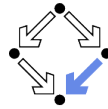
Markers */\*@..* indicate hidden mathematical annotations.

Wolfgang Schreiner

<http://www.risc.jku.at>

24/46

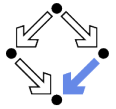
## A Theory



```
/*@
theory {
  sum: (INT, INT) -> INT;
  sumaxiom: AXIOM
  FORALL(m: INT, n: INT):
    IF n < m THEN
      sum(m, n) = 0
    ELSE
      sum(m, n) = n + sum(m, n-1)
    ENDIF;
}
@*/
class Sum
...
```

The introduction of a function  $sum(m, n) = \sum_{j=m}^n j$ .

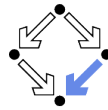
## A Method Specification



```
static int sum(int n) /*@
  requires VAR n < Base.MAX_INT;
  ensures
    LET result=VALUE@NEXT IN
    IF VAR n < 0
      THEN result = -1
    ELSE result = sum(1, VAR n)
  ENDIF;
@*/
...
```

For non-negative  $n$ , a call of program method  $sum(n)$  returns  $sum(1, n)$  (and does not modify any global variable).

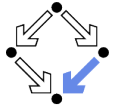
## A Loop Annotation



```
while (i <= n) /*@
  invariant VAR n < Base.MAX_INT
    AND 1 <= VAR i AND VAR i <= VAR n+1
    AND VAR s=sum(1, VAR i-1);
  decreases VAR n - VAR i + 1;
@*/
{
  s = s+i;
  i = i+1;
}
}
```

The loop invariant and termination term (measure).

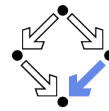
## The Specification Language



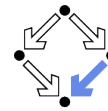
Derived from the language of the RISC ProofNavigator.

- **State conditions/relations, state terms.**
  - State condition: method precondition (requires).
  - State relation: method postcondition (ensures), loop invariant (invariant).
  - State term: termination term (decreases).
- **References to program variables.**
  - OLD  $x$ : the value of program variable  $x$  in the pre-state.
  - VAR  $x$ : the value of program variable  $x$  in the post-state.
  - In state conditions/terms, both refer to the value in the current state.
  - If program variable is of the program type  $T$ , then then OLD/VAR  $x$  is of the mathematical type  $T'$ .
    - $\text{int} \rightarrow \text{Base.int} = [\text{Base.MIN\_INT}, \text{Base.MAX\_INT}]$ .
- **Function results**
  - VALUE@NEXT: the return value of a program function.
  - The value of the function call's post-state NEXT.

# The Semantics View



# The Method Body



## Body Knowledge

[Show Original Formulas]

## Pre-State Knowledge

$old\ n < Base.MAX_{INT}$

## Effects

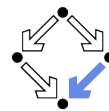
**executes:** false, **continues:** false, **breaks:** false, **returns:** true  
**variables:** -; **exceptions:** -

## Transition Relation

```
if old n < 0 then
  returns@next ∧ value@next = -1
else
  returns@next
  ∧
  (∃in ∈ Base.int: in = old n + 1 ∧ 1 ≤ in ∧ value@next = sum(1, in - 1))
  ∧
  old n < Base.MAX_{INT}
endif
```

Select method symbol "sum" and menu entry "Show Semantics".

# A Body Command



## Statement Knowledge

[Show Original Formulas]

## Pre-State Knowledge

$old\ n < Base.MAX_{INT} \wedge old\ n \geq 0 \wedge old\ s = 0 \wedge old\ i = 1$

## Precondition

$old\ n < Base.MAX_{INT} \wedge 1 \leq old\ i \wedge old\ i \leq old\ n + 1 \wedge old\ s = sum(1, old\ i - 1)$

## Effects

**executes:** true, **continues:** false, **breaks:** false, **returns:** false  
**variables:** s, i; **exceptions:** -

## Transition Relation

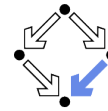
$var\ i = old\ n + 1 \wedge old\ n < Base.MAX_{INT} \wedge 1 \leq var\ i \wedge var\ s = sum(1, var\ i - 1)$

## Termination Condition

$executes@now \Rightarrow old\ n - old\ i \geq -1$

Move the mouse pointer over the box to the left of the loop.

# The Semantics Elements



- **Pre-State Knowledge**

What is known about the pre-state of the command.

- **Precondition**

What has to be true for the pre-state of the command such that the command may be executed.

- **Effects**

Which kind of effects may the command have.

- **variables:** which variables may be changed.

- **exceptions:** which exceptions may be thrown.

- **executes, continues, breaks, returns:** may the execution terminate normally, may it be terminated by a continue, break, return.

- **Transition Relation**

The prestate/poststate relationship of the command.

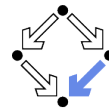
- **Termination**

What has to be true for the pre-state of the command such that the command terminates.

Formulas are shown after simplification (see "Show Original Formulas").



## Constraining a State



Select the loop body, enter in the box the condition  $\text{VAR } s=2 \text{ AND } \text{VAR } i=1$ , press "Submit", and move the mouse to  $i=i+1$ .

### State Conditions

[Show Original Formulas]

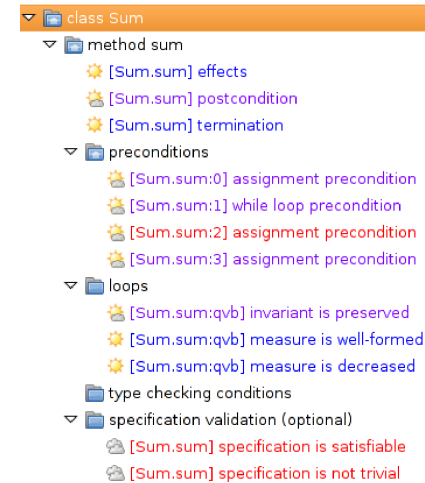
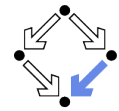
### Pre-State Condition

$\text{var } i = 1 \wedge \text{var } s = \text{var } i + 2$

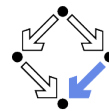
### Post-State Condition

$\text{var } s = 3 \wedge \text{var } i = 2$

## The Verification Tasks



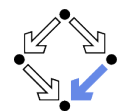
## The Verification Tasks



- **Effects:** does the method only change those global variables indicated in the method's assignable clause?
- **Postcondition:** do the method's precondition and the body's state relation imply the method's postcondition?
- **Termination:** does the method's precondition imply the body's termination condition?
- **Precondition:** does a statement's prestate knowledge imply the statement's precondition?
- **Loops:** is the loop invariant preserved, the measure well-formed (does not become negative) and decreased?
- **Type checking conditions:** are all formulas well-typed?
- **Specification validation:** does for every input that satisfies a precondition exist a result that does (not) satisfy the postcondition?

Partially solved by automatic decision procedure, partially by an interactive computer-supported proof.

## The Task States

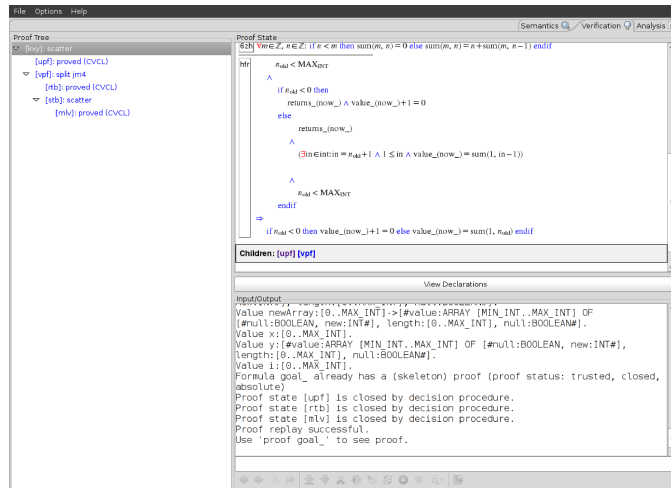
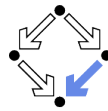


The task status is indicated by color (icon).

- **Blue (sun):** the task was solved in the current execution of the RISC ProgramExplorer (automatically or by an interactive proof).
- **Violet (partially clouded):** the task was solved in a previous execution by an interactive proof.
  - Nothing has changed, so we need not perform the proof again.
  - However, we may replay the proof to investigate it.
- **Red (partially clouded):** there exists a proof but it is either not complete or cannot be trusted any more (something has changed).
- **Red (fully clouded):** there does not yet exist a proof.

Select "Execute Task" to start/replay a proof, "Show Proof" to display a proof, "Reset Task" to delete a proof.

## A Postcondition Proof

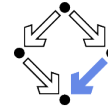


Wolfgang Schreiner

<http://www.risc.jku.at>

37/46

## Linear Search



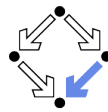
```
/*@..
public class Searching
{
  public static int search(int[] a, int x) /*@..
  {
    int n = a.length;
    int r = -1;
    int i = 0;
    while (i < n && r == -1) /*@..
    {
      if (a[i] == x)
        r = i;
      else
        i = i+1;
    }
    return r;
  }
}
```

Wolfgang Schreiner

<http://www.risc.jku.at>

38/46

## The Representation of Arrays



The program type `int []` is mapped to the mathematical type `Base.IntArray`.

```
theory Base
{
  ...
  IntArray: TYPE =
    [#value: ARRAY int OF int, length: nat, null: BOOLEAN#];
  ...
}
```

- `(VAR a).length`: the number of elements in array `a`.
- `(VAR a).value[i]`: the element with index `i` in array `a`.
- `(VAR a).null`: `a` is the null pointer.

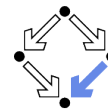
Program type `Class` is mapped to mathematical type `Class.Class`;  
`Class []` is mapped to `Class.Array`.

Wolfgang Schreiner

<http://www.risc.jku.at>

39/46

## Theory



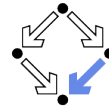
```
/*@
theory uses Base {
  int: TYPE = Base.int;
  intArray: TYPE = Base.IntArray;
  smallestPosition: FORMULA
  FORALL(a: intArray, n: NAT, x: int):
    (EXISTS(i:int): 0 <= i AND i < n AND a.value[i] = x) =>
    (EXISTS(i:int): 0 <= i AND i < n AND a.value[i] = x AND
     (FORALL(j:int): 0 <= j AND j < n AND a.value[j] = x =>
      j >= i));
}
@*/
public class Searching
...
```

Wolfgang Schreiner

<http://www.risc.jku.at>

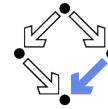
40/46

## Method Specification



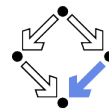
```
public static int search(int[] a, int x) /*@
  requires (VAR a).null = FALSE;
  ensures
    LET result = VALUE@NEXT, n = (VAR a).length IN
    IF result = -1 THEN
      FORALL(i: INT): 0 <= i AND i < n =>
        (VAR a).value[i] /= VAR x
    ELSE
      0 <= result AND result < n AND
      (FORALL(i: INT): 0 <= i AND i < result =>
        (VAR a).value[i] /= VAR x) AND
      (VAR a).value[result] = VAR x
    ENDIF;
  @*/
  ...
```

## Loop Annotation



```
while (i < n && r == -1) /*@
  invariant (VAR a).null = FALSE AND VAR n = (VAR a).length
    AND 0 <= VAR i AND VAR i <= VAR n
    AND (FORALL(i: INT): 0 <= i AND i < VAR i =>
      (VAR a).value[i] /= VAR x)
    AND (VAR r = -1 OR (VAR r = VAR i AND VAR i < VAR n AND
      (VAR a).value[VAR r] = VAR x));
  decreases IF VAR r = -1 THEN VAR n - VAR i ELSE 0 ENDIF;
  @*/
{
  if (a[i] == x)
    r = i;
  else
    i = i+1;
}
```

## Method Semantics



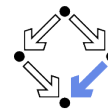
### Transition Relation

```
(∃ in ∈ Base.int, n ∈ Base.int:
  n = old a.length ∧ (in ≥ n ∨ value@next ≠ -1) ∧ 0 ≤ in ∧ in ≤ n
  ∧
  (∀ i ∈ ℤ: 0 ≤ i ∧ i < in ⇒ old a.value[i] ≠ old x)
  ∧
  ( value@next = -1
  ∨
  value@next = in ∧ in < n ∧ old a.value[value@next] = old x) ) ∧ ¬ old a.null
  ∧
  returns@next
```

### Termination Condition

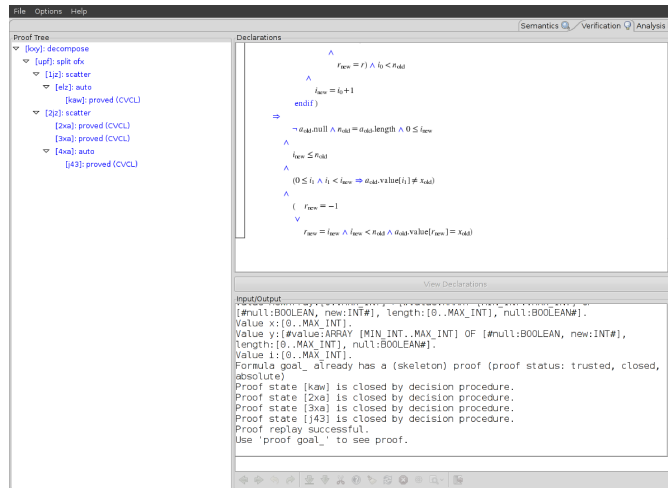
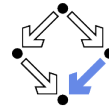
```
executes@now ⇒ old a.length ≥ 0
```

## Verification Tasks

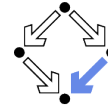


- method search
  - [Searching.search] effects
  - [Searching.search] postcondition
  - [Searching.search] termination
  - preconditions
    - [Searching.search:0] declaration precondition
    - [Searching.search:1] declaration precondition
    - [Searching.search:2] while loop precondition
    - [Searching.search:3] conditional precondition
    - [Searching.search:4] assignment precondition
  - loops
    - [Searching.search:rb] invariant is preserved
    - [Searching.search:rb] measure is well-formed
    - [Searching.search:rb] measure is decreased
  - type checking conditions
    - [Searching.(local):p3x] value is in interval
    - [Searching.(local):smu] value is in interval
    - [Searching.(local):unx] value is in interval
  - specification validation (optional)

# Invariant Proof



# Working Strategy



- Develop theory.
  - Introduce interesting theorems that may be used in verifications.
- Develop specifications.
  - Validate specifications, e.g. by showing satisfiability and non-triviality.
- Develop program with annotations.
  - Validate programs/annotations by investigating program semantics.
- Prove postcondition and termination.
  - Partial and total correctness.
  - By proofs necessity of additional theorems may be detected.
- Prove precondition tasks and loop tasks.
  - By proofs necessity of additional theorems may be detected.
- Prove mathematical theorems.
  - Validation of auxiliary knowledge used in verifications.

The integrated development of theories, specifications, programs, annotations is crucial for the design of provably correct programs.