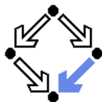
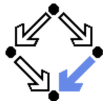
The RISC ProgramExplorer
A First Status Report

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at



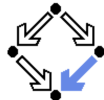


1. Background

2. Programming and Specification Language

3. The Software

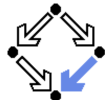
Goals



An integrated program reasoning environment that provides insight into the semantic essence of a program.

- Is based on the concept of programs as state relations.
  - A program implements a relation on states.
  - A specification describes a relation on states.
  - The program relation must imply the specification relation.
- Addresses various semantic questions.
  - Is a specification trivial or not implementable?
  - What is the state relation described by a command/method?
  - What state condition is known at a particular program point?
  - Are methods only called in states that satisfy the methods' preconditions?
  - Assuming that loop invariants hold and termination terms for loops and recursive methods are appropriate, does the method meet its specification?
  - Do the invariants indeed hold?
  - Are the termination terms indeed appropriate?
- Provides a state-of-the-art graphical user interface.
  - Tight links between syntactic source code and semantic essence.
  - Helps to gain insight as much as possible.

History



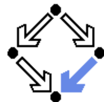
- 2005-2006: The RISC ProofNavigator.
  - Computer-aided proving assistant based on an SMT solver (CVCL) resulting from previous experience with PVS, Isabelle, Coq, ...
  - Proving conditions that are manually derived from the verification of sequential programs and concurrent systems.

    Wolfgang Schreiner. *The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom*. Formal Aspects of Computing, Springer, April 2008.
- 2006-2008: Programs as State Relations.
  - Program semantics, specification semantics, reasoning calculus, soundness proof of calculus with respect to semantics.
  - Consideration of control flow interruptions and undefined expressions.

    Wolfgang Schreiner. *A Program Calculus*. Technical Report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2008.
- 2008–: The RISC ProgramExplorer.
  - Software framework prepared to accommodate the calculus.

    Technical report to be prepared.

The RISC ProofNavigator



RISC ProofNavigator

File Options Help

Proof Tree

- [dca]: expand Invariant, Output
  - [tvy]: scatter
    - [dcu]: auto
      - [t4c]: proved (CVCL)
    - [ecu]: split pkg
      - [kel]: proved (CVCL)
      - [lel]: scatter
        - [lvn]
    - [fcu]
    - [gcu]: proved (CVCL)

Proof State

Formula [C] proof state [lvn]

Constants (with types): anyelem, r, get, length, put, Invariant, content, j_0, anyarray, new, Output, Input, oldx, i, a, n, olda, any, x.

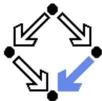
ed2	$\text{olda} = a$
cmz	$\text{oldx} = x$
hvv	$n = \text{length}(a)$
564	$\forall j \in \mathbb{N}: x = \text{get}(a, j) \Rightarrow j \geq i$
mys	$i \leq n$
gkr	$r = -1 \vee r = i \wedge x = \text{get}(a, r) \wedge i < n$
orv	$r = -1 \Rightarrow n \leq i$
k4w	$x = \text{get}(a, j_0)$
6ha	$j_0 < n$
jh5	$0 \leq r$

View Declarations

Input/Output

```
ELEM], Invariant: ([NAT, ARRAY NAT OF ELEM], ELEM, NAT, NAT, INT)->BOOLEAN, content: [NAT,
ARRAY NAT OF ELEM]->ARRAY NAT OF ELEM, j_0: NAT, anyarray: [NAT, ARRAY NAT OF ELEM], new: NAT-
>[NAT, ARRAY NAT OF ELEM], Output: BOOLEAN, Input: BOOLEAN, oldx: ELEM, i: NAT, a: [NAT,
ARRAY NAT OF ELEM], n: NAT, olda: [NAT, ARRAY NAT OF ELEM], any: ARRAY NAT OF ELEM, x: ELEM.
[ed2] olda = a
[cmz] oldx = x
[hvv] n = length(a)
[564] FORALL(j:NAT): x = get(a, j) => j >= i
[mys] i <= n
[gkr] r = -1 OR r = i AND x = get(a, r) AND i < n
[orv] r = -1 => n <= i
[k4w] x = get(a, j_0)
[6ha] j_0 < n
------------------------------------------------------------
[jh5] 0 <= r
prove>
```

Programs as State Relations



- Hoare calculus: two formulas.

  $\{x = a\}$ `x=x+1` $\{x = a + 1\}$

- Dynamic logic: one formula with modalities.

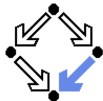
  $\forall a : x = a \Rightarrow$ [`x=x+1`] $x = a + 1$

- Relational approach: one formula with primed variables.

  `x=x+1`: $x' = x + 1$

Core idea: translate programs to state relations described by formulas in (essentially) classical predicate logic with classical rules of reasoning.

Example



- Program

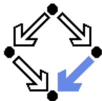
```
x=x+1; if (x == 0) return 1; else y = x*x
```

- Formula

```
x' = ADD32(x,1) AND
IF x' = 0
  THEN next.returns AND next.value = 1 AND y' = y
  ELSE next.executes AND y' = MULT32(x,x)
```

Effect of command fully described in a classical logical framework.

Semantics of Commands



Take command C, context c, states s and s'.

- Transition relation $[\![_]\!]$:

  $$[\![C]\!]^c(s, s') \Leftrightarrow \ldots$$

  - $[\![C]\!]^c$ defines a relation on states.
  - Which state transitions are possible by execution of C in c?
- Termination condition $\langle\!\langle _ \rangle\!\rangle$:

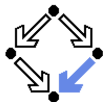
  $$\langle\!\langle C \rangle\!\rangle^c(s) \Leftrightarrow \ldots$$

  - $\langle\!\langle C \rangle\!\rangle^c$ defines a condition on states.
  - For which prestates must the execution of C yield a poststate?
- Constraint:

  $$\forall s : \langle\!\langle C \rangle\!\rangle^c(s) \Rightarrow \exists s' : [\![C]\!]^c(s, s').$$

A command is translated to a state relation and a state condition.

Semantics of Formulas



Take logical formula F, context c, environment e, states s and s'

- Binary formula semantics $[\![\,\textvisiblespace\,]\!]$:

  $$[\![\, F \,]\!]^c_e(s, s') \Leftrightarrow \ldots$$

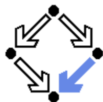
  - $[\![\, F \,]\!]^c_e$ defines a relation on states.
- Unary formula semantics $[\![\,\textvisiblespace\,]\!]$:

  $$[\![\, F \,]\!]^c_e(s) \Leftrightarrow [\![\, F \,]\!]^c_e(s, s)$$

  - $[\![\, F \,]\!]^c_e$ defines a condition on states.

State relations and state conditions can be specified in classical logic.

Reasoning Calculus



Various kinds of judgements that describe properties of commands.

$\ldots \vdash C : F \Leftrightarrow$
$\forall c, e, s, s' : \ldots \Rightarrow$
$[\![C]\!]^c(s, s') \Rightarrow [\![F]\!]^c_e(s, s')$

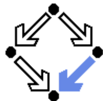
$\ldots \vdash C \downarrow^I F \Leftrightarrow$
$\forall c, s, e : \ldots \Rightarrow$
$[\![F]\!]^c(s) \Rightarrow \langle\!\langle C \rangle\!\rangle^c_e(s)$

$\ldots \vdash C \checkmark F \Leftrightarrow$
the execution of C in a state that satisfies F
does not encounter undefined expressions

$\ldots$

We have a calculus for deriving only true judgements.

Theoretical Framework

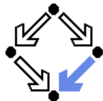


Formal syntax and semantics of various languages.

- An abstract imperative programming language.
  - Commands operating on states.

    `=`, `var`, `if`, `while`, `continue`, `break`, `return`, `throw`, `try`.
  - Methods with results, (direct and indirect) recursion.
- An abstract logic formula language.
  - Predicate logic formulas with functions and predicates on states.
- A program specification language based on the formula language.
  - Assertions, loop invariants, termination terms.
  - Method specifications with preconditions, postconditions, frame conditions, exception conditions, recursion measures.

The formal reasoning calculus was elaborated and its soundness was proved within this framework.

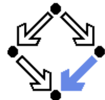


1. Background

2. Programming and Specification Language

3. The Software

The Concrete Programming Language



A subset of Java ("MiniJava") that can be mapped to the abstract programming language in a rather straight-forward way.

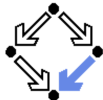
- Classes as modules with class variables and class methods.
  - Treatment as global variables and methods of the basic calculus.
- Classes as types with object variables, constructors, object methods.
  - Object functions receive the `this` object as an additional argument and return it as an additional result.
- Value semantics for arrays and objects.

  Type checker prevents aliasing (i.e. that different variables refer to same object) and thus hides difference to reference semantics.

  - Assignment to variable only from a constructor call.
  - Return as function result only from locally owned object.
  - Passing as an argument only from a constructor call or from a local variable that does not appear as another argument.
  - No (directly or indirectly) recursive class references.

Classes as modules and types, no inheritance, no reference semantics.

Example

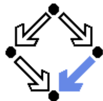


```
class Record {
  String key; int value;
  Record(String k, int v)  { key = k; value = v; }
  boolean equals(String k) { boolean e = key.equals(k); return e; }

  public static int search(Record[] a, String key) {
    int n = a.length;
    for (int i=0; i<n; i++) {
      Record r = new Record(a[i].key, a[i].value); // copy of a[i]
      boolean e = r.equals(key);                   // a[i].equals(key) illegal
      if (e) return i;
    }
    return -1;
  }

  public static void main() {
    Record[] a = new Record[10];
    for (int i=0; i<10; i++) a[i] = new Record("abc", i);
    int i = search(a, "abc");
    System.out.println(i);
  }
}
```

The Concrete Specification Language

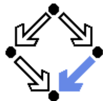


- Typed higher-order predicate logic.
  - ProofNavigator syntax (inherited from CVS/PVS).

    ```
    FORALL(i:INT): 0 <= i AND i < n => a0[i].key /= k0
    ```
- Program variables.
  - $x \rightsquigarrow$ `old x`, $x' \rightsquigarrow$ `var x`.
- State types, constants functions, predicates.
  - `STATE(`*T*`)`, `now`, `next`, `executes@`*s*, `value@`*s*, ...
- Method specifications
  - `assignable` ... `signals` ... `requires` ... `ensures` ... `decreases`
- Code annotations
  - Loops: `invariant` ... `decreases` ...
  - Statements: `assert` ...

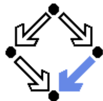
Tradition of JML et al, extended by an explicit notion of states.

Example



```
public static int search(Record[] a, String key) /*@
  requires var a /= Record.nullArray;
  ensures executes@next AND
    (LET result=value@next, a0=var a, n=Record.length(a0), k0=var key IN
      IF result = -1 THEN
        FORALL(i:INT): 0 <= i AND i < n => a0[i].key /= k0
      ELSE
        0 <= result AND result < n AND a0[result].key = k0
      ENDIF);
@*/
{
  int n = a.length;
  for (int i=0; i<n; i++)
  {
    Record r = new Record(a[i].key, a[i].value);
    boolean e = r.equals(key);
    if (e) return i;
  }
}
```

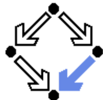
Theories



- Automatically generated theories.
  - `class` C $\rightsquigarrow$ `theory` C.
  - Classes as types.
- Named theories
  - File *Theory*`.theory`.
  - Abstract datatypes etc.
- Local theories.
  - `/*@ theory { ...} @*/ class` C
  - Local definitions inside a class.

Building blocks for specifications.

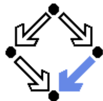
Example



```
theory Record uses java.lang.String, Base { // generated from class Record
  Record: TYPE = [#key: java.lang.String.String, value: Base.int#];
  null: Record; nullArray: ARRAY Base.int OF Record;
  length: (ARRAY Base.int OF Record) -> Base.nat;
}

theory Stack {                              // file Stack.theory
  Elem: TYPE = INT; Stack: TYPE;
  empty: Stack; cons: (Elem, Stack) -> Stack;
  isempty: PREDICATE(Stack);
  IE: AXIOM FORALL(s: Stack): isempty(s) <=> s=empty;
}

/*@
theory uses Record, java.lang.String {      // file Record.java
  Record: TYPE = Record.Record;
  String: TYPE = java.lang.String.String;
  notFound: PREDICATE(ARRAY INT OF Record, INT, STRING) =
    PRED(a:ARRAY INT OF Record, i:INT, key: String):
      (FORALL(i:INT): 0 <= i AND i < Record.length(a) => a[i].key /= key);
} @*/
class Record {...}
```

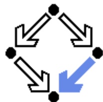


1. Background

2. Programming and Specification Language

3. **The Software**

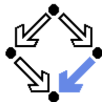
The Software



- Integrated environment built on top of the Eclipse SWT.
  - Provides graphical user interface and editing framework.
- Analyze view.
  - Source code editor.
    - Syntax highlighting.
    - Specification text folding.
    - Error annotations.
    - Active identifiers.
  - Console.
  - Files.
  - Symbols.
  - Tasks.
    - Symbols and tasks linked to source code.
- Verify view.
  - Embedding of the RISC ProofNavigator.

More views to be added on demand.

The Software



RISC ProgramExplorer

File Edit Help

Verify Analyze

Files | Symbols

- ProgramExplorer
- ProofNavigator
- RCS
- java
- Übung
- Basic
- Basic0
- Demonstrator
- Main
- Record
  - (local)
  - key
  - value
  - Record(java.lang.St
  - main()
  - search(Record[], ja
  - equals(java.lang.Str
- Test
- Base
- Basic
- Basic0
- Demonstrator
- List
- Main
- Queue
- Record
- Stack
- Test

Record.java

```
1 /*@
10 class Record
11 {
12   String key; int value;
13   Record(String k, int v)  { key = k; value = v; }
14   boolean equals(String k) { boolean e = key.equals(k); return e;
15
16   public static int search(Record[] a, String key) /*@
17     requires var a /= Record.nullArray;
18     ensures executes@next AND
19       (LET result=value@next, a0=var a, n=Record.length(a0), k0=va
20         IF result = -1 THEN
21           FORALL(i:INT): 0 <= i AND i < n => a0[i].key /= k0
22         ELSE
23           0 <= result AND result < n AND a0[result].key = k0
24         ENDIF);
25   @*/
26   {
27     int n = a.length;
28     for (int i=0; i<n; i++)
29     {
30       Record r = new Record(a[i].key, a[i].value);
31       boolean e = r.equals(key);
32       if (e) return i;
33     }
```

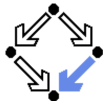
Console

```
class Record was processed with no errors
theory Record was processed with no errors
class Record was processed with no errors
theory Record was processed with no errors
class Record was processed with no errors
theory Record was processed with no errors
theory List (file /usr2/schreine/projects/FM-RISC2/test/List.theory)
class Record (file /usr2/schreine/projects/FM-RISC2/test/Record.java)
class Basic was processed with no errors
theory Basic was processed with no errors
class Record was processed with no errors
theory Record was processed with no errors
```

All Tasks | Open Tasks

- class Array
- class Basic
- class Basic0
- class Demonstrator
- class Main
- class Record
  - method Record
  - method equals
  - method main
  - method search
    - type checking conditions
      - [ivg] value is in interval
  - theory (local)
  - type checking conditions
- class Test
- package ProgramExplorer
- package ProofNavigator
- package RCS
- package java
- package Übung
- theory Base
  - type checking conditions
    - [2f4] Interval [MIN_INT..MAX_
    - [xlq] Interval [0..MAX_INT] is
- theory Basic
- theory Basic0
- theory Demonstrator
- theory List

Core Functionality

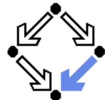


Constructing/maintaining the internal model of the program/specification.

- Annotated abstract syntax trees.
  - Nodes linked to source code positions.
  - Identifiers linked to symbols.
  - Terms linked to types.
- Symbol tables.
  - Collections of symbols introduced in same scope.
  - Symbols linked to abstract syntax tree nodes.
- Proving tasks.
  - Organized in nested folders, linked to abstract syntax tree nodes.
  - Currently: type-checking tasks.
  - Later: various reasoning tasks.

Parsing, type checking, semantic processing; linking source code to model and vice versa; propagating information from model to source code.

Internal Model



Source Code

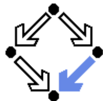
Abstract Syntax Tree

Symbol Table

Tasks

Tight integration of the various elements.

Task Management

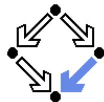


Framework for generation and maintenance of tasks.

- Tasks organized in nested folders.
  - Corresponding to source code structure.
  - Linked to source code positions.
- Strategies may be associated to tasks.
  - Currently: automatic decision by CVCL and manual verification.
- Tasks may be translated to proving problems.
  - Type-checking task → state logic problem → classical logic problem → RISC ProofNavigator problem.
- Proofs are persistent.
  - Stored in RISC ProofNavigator format.
  - Reused in new RISC ProgramExplorer invocations.
  - RISC ProofNavigator dependence control maintains trust status.

Subsequent reasoning/verification tasks will be built upon this framework.

Generated Proving Problem



RISC ProgramExplorer

File Options Help

Verify Analyze

Proof Tree

[kxy]: proved (CVCL)

Proof State

Formula [goal_] proof state [kxy] (autosimp): proved (CVCL)

Constants (with types): breaks, lengthBooleanArray, nullArray$_0$, returns, nullStringArray, MIN_{INT}, nullCharArray, throwsException, old_n, length, message, lengthIntArray, next, executes, nullBooleanArray, null, literal0, throws, lengthStringArray, nullArray, now, MAX_{INT}, lengthCharArray, i, continues, value, length$_0$, s, nullIntArray, r, fac, null$_0$, x.

Axioms: $x \geq 0$.

oar

$$\text{old_n} > 0 \wedge \text{value}(\text{next}) = 0 \wedge \text{old_n} \neq 0 \wedge \text{value}(\text{now}) = \frac{1}{\text{old_n}} \wedge \text{old_n} > 0 \Rightarrow MIN_{INT} \geq 0 \vee \text{old_n} \geq 0$$

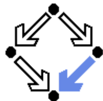
Children:

View Declarations

Input/Output

```
Type STATE.
Value _next:STATE.
Value _now:STATE.
Value _value:STATE->[MIN_INT..MAX_INT].
Value _message:STATE->STRING.
Value _breaks:STATE->BOOLEAN.
Value _continues:STATE->BOOLEAN.
Value _executes:STATE->BOOLEAN.
Value _returns:STATE->BOOLEAN.
Value _throws:STATE->BOOLEAN.
Value _throwsException:(STATE, INT)->BOOLEAN.
Value _old_n:[MIN_INT..MAX_INT].
Proof read (proof status: trusted, closed, absolute).
```

Current State and Further Work



- Software in alpha status.
  - Reasonably stable (tested with toy examples only).
  - Classes: ca. 80 ProgramExplorer, 100 ProofNavigator, 300 syntax.
  - Lines of code: about 116K with comments (perhaps 40-50K without).
- User manual.
  - Documenting the languages and the software.
- Reasoning calculus.
  - Integration of the various kinds of formal judgements.
  - Generation of the various kinds of verification conditions.

Next prototype with some elements of the calculus by the end of 2010.