Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs[*]

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University
Linz, Austria
Wolfgang.Schreiner@risc.jku.at

Extended Abstract

Most systems for program reasoning are based on calculi such as the Hoare Calculus or Dynamic Logic [1] where we generate from a program specification and a program implementation (which is annotated with additional meta-information such as loop invariants and termination terms) those conditions whose verification implies that the implementation indeed meets the specification. The problem is that by such an approach we gain little insight into the program before respectively independently of the verification process. In particular, if the verification attempt is a priori doomed to fail because of errors, inconsistencies, or weaknesses in the program's specification, implementation, or meta-information (which is initially the case in virtually all verification attempts), we will learn so only by unsuccessfully struggling with the verification until some mental "click" occurs. This click occurs frequently very late, because, in the heat of the struggle, it is usually hard to see whether the inability to perform a correctness proof is due to an inadequate proving strategy or due to errors or inconsistencies in the program. Actually, it is usually the second factor that contributes most to the time spent and frustration experienced; once we get the specification/implementation/meta-information correct, the verification is a comparatively small problem. We have frequently observed this fact in our own verification attempts as well as in those performed by students of computer science and mathematics in courses on formal methods.

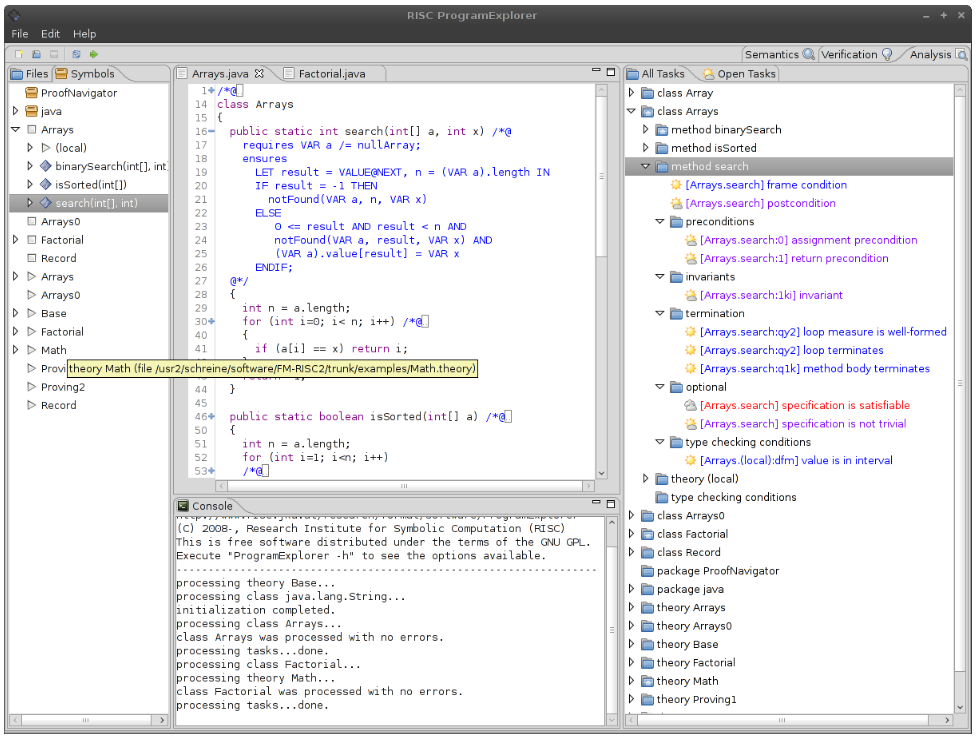
We therefore advocate an alternative approach where we insert between a program and its verification conditions an additional layer, the denotation of the program [4] expressed in a declarative form. The program (annotated with its meta-information) is translated into its denotation from which subsequently the verification conditions are generated. However, even before (and independently of) any verification attempt, one may investigate the denotation itself to get insight into the "semantic essence" of the program, in particular to see whether the denotation indeed gives reason to believe that the program has the expected behavior. Errors in the program and in the meta-information may thus be detected and fixed prior to actually performing the formal verification.

More concretely, following the relational approach to program semantics [2], we model the effect of a program (command) c as a binary relation $[\![c]\!]$ on program states which describes the possible pairs of pre- and post-states of c. Such a relation can be also described in a declarative form by a logic formula f_r with denotation $[\![f_r]\!]$. Thus a formal calculus is devised to derive from a program c a judgment $c : f_r$ such that $[\![c]\!] \subseteq [\![f_r]\!]$. For instance, we can derive

$$\texttt{x=x+1} : \mathsf{var}\ x = \mathsf{old}\ x + 1$$

[*]Sponsored by the Austrian Science Fund (FWF) in the frame of the DK "Computational Mathematics" (W1214).

Submitted to:
THedu'11



```
ProofNavigator
java
Arrays
  (local)
  binarySearch(int[], int
  isSorted(int[])
  search(int[], int)
Arrays0
Factorial
Record
Arrays
Arrays0
Base
Factorial
Math
Provi
Proving2
Record
```

theory Math (file /usr2/schreine/software/FM-RISC2/trunk/examples/Math.theory)

```
/*@
class Arrays
{
  public static int search(int[] a, int x) /*@
    requires VAR a /= nullArray;
    ensures
      LET result = VALUE@NEXT, n = (VAR a).length IN
      IF result = -1 THEN
        notFound(VAR a, n, VAR x)
      ELSE
         0 <= result AND result < n AND
         notFound(VAR a, result, VAR x) AND
         (VAR a).value[result] = VAR x
      ENDIF;
  @*/
  {
    int n = a.length;
    for (int i=0; i< n; i++) /*@
    {
      if (a[i] == x) return i;
  }

  public static boolean isSorted(int[] a) /*@
  {
    int n = a.length;
    for (int i=1; i<n; i++)
    /*@
```

```
(C) 2008-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "ProgramExplorer -h" to see the options available.
-----------------------------------------------------------------
processing theory Base...
processing class java.lang.String...
initialization completed.
processing class Arrays...
class Arrays was processed with no errors.
processing tasks...done.
processing class Factorial...
processing theory Math...
class Factorial was processed with no errors.
processing tasks...done.
```

```
class Array
class Arrays
  method binarySearch
  method isSorted
  method search
    [Arrays.search] frame condition
    [Arrays.search] postcondition
    preconditions
      [Arrays.search:0] assignment precondition
      [Arrays.search:1] return precondition
    invariants
      [Arrays.search:1ki] invariant
    termination
      [Arrays.search:qy2] loop measure is well-formed
      [Arrays.search:qy2] loop terminates
      [Arrays.search:q1k] method body terminates
    optional
      [Arrays.search] specification is satisfiable
      [Arrays.search] specification is not trivial
    type checking conditions
      [Arrays.(local):dfm] value is in interval
  theory (local)
  type checking conditions
class Arrays0
class Factorial
class Record
package ProofNavigator
package java
theory Arrays
theory Arrays0
theory Base
theory Factorial
theory Math
theory Proving1
```

Figure 1: The RISC ProgramExplorer

where the logic variable old x refers to the value of the program variable x in the prestate of the command and the logic variable var x refers to its value in the poststate. In this way, we can constrain the allowed state transitions, i.e. handle the partial correctness of programs. To capture also total correctness, we introduce the set of states $\langle\!\langle c \rangle\!\rangle$ on which the execution of c must terminate ($\langle\!\langle c \rangle\!\rangle$ is a subset of the domain of $[\![c]\!]$). Such a set can be also described in a declarative form by a logic formula (a state condition) f_c. Thus we derive a judgment $c \downarrow f_c$ such that $[\![f_c]\!] \subseteq \langle\!\langle c \rangle\!\rangle$. In this fashion, the pair of formulas f_r and f_c captures the semantic essence of c in a declarative form that is open for inspection and manipulation.

We have implemented this idea in a comprehensive form in the *RISC ProgramExplorer*[1], a new program reasoning environment for educational purposes which encompasses the previously developed *RISC ProofNavigator* as an interactive proving assistant [5]. The RISC ProgramExplorer supports reasoning about programs written in a restricted form of Java (including support for control flow interruptions such as `continue`, `break`, `return`, and `throw`, static and dynamic methods, classes and a restricted form of objects) and specified in the formula language of the RISC ProofNavigator (which is derived from PVS [3]). The system is currently in beta state, a first release under the GNU Public License will be available by July 2011 and will be subsequently used in regular courses. A screenshot of the software is given in Figure 1; in the remainder of this abstract, we will first sketch the formalism on which the RISC ProgramExplorer is based and then give an illustrative example.

For the purpose of this presentation, we use a simple command language without control flow interruptions and method calls; a command c can be formed according to the grammar

$$c ::= x = e \mid \{\texttt{var}\ x;\ c\} \mid \{c_1; c_2\} \mid \texttt{if}\ (e)\ \texttt{then}\ c \mid \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ (e)^{f,t}\ c$$

[1]`http://www.risc.jku.at/research/formal/software/ProgramExplorer`

$$\frac{c : [f]^{xs}_{g,h} \qquad x \notin xs}{c : [f \land \mathsf{var}\ x = \mathsf{old}\ x]^{xs\, \cup\, \{x\}}_{g,h}} \qquad \frac{e \simeq_h t}{x = e : [\mathsf{var}\ x = t]^{\{x\}}_{\mathsf{true},h}} \qquad \frac{c : [f]^{xs}_{g,h}}{\{\mathtt{var}\ x;\ c\} : [\exists x : f]^{xs\setminus x}_{g,\forall x : h[x/\mathsf{old}\ x]}}$$

$$\frac{c_1 : [f_1]^{xs}_{g_1,h_1} \qquad c_2 : [f_2]^{xs}_{g_2,h_2} \qquad \mathrm{PRE}(c_2,h_2) = h_3}{\{c_1\,;c_2\} : [\exists ys : f_1[ys/\mathsf{var}\ xs] \land f_2[ys/\mathsf{old}\ xs]]^{xs}_{g_1 \land g_2, h_1 \land h_3}}$$

$$\frac{e \simeq_h f_e \qquad c_1 : [f_1]^{xs}_{g_1,h_1}}{\mathtt{if}\ (e)\ \mathtt{then}\ c : [\mathsf{if}\ f_e\ \mathsf{then}\ f_1\ \mathsf{else}\ \mathsf{var}\ xs = \mathsf{old}\ xs]^{xs}_{g_1,h \land (f_e \Rightarrow h_1)}}$$

$$\frac{e \simeq_h f_e \qquad c_1 : [f_1]^{xs}_{g_1,h_1} \qquad c_2 : [f_2]^{xs}_{g_2,h_2}}{\mathtt{if}\ (e)\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : [\mathsf{if}\ f_e\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2]^{xs}_{g_1 \land g_2, h \land \mathsf{if}\ f_e\ \mathsf{then}\ h_1\ \mathsf{else}\ h_2}}$$

$$\frac{\begin{array}{c} e \simeq_h f_e \qquad c : [f_c]^{xs}_{g_c,h_c} \\ g \equiv \forall xs, ys, zs : f[xs/\mathsf{old}\ xs, ys/\mathsf{var}\ xs] \land f_e[ys/\mathsf{old}\ xs] \land f_c[ys/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \Rightarrow \\ h[ys/\mathsf{old}\ xs] \land f[xs/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \end{array}}{\mathtt{while}\ (e)^{f,t}\ c : [f \land \neg f_e[\mathsf{var}\ xs/\mathsf{old}\ xs]]^{xs}_{g_c \land g, h \land f[\mathsf{old}\ xs/\mathsf{var}\ xs]}}$$

Figure 2: The Transition Rules

$$x = e \downarrow_{\mathsf{true}} \mathsf{true} \qquad \frac{c \downarrow_g f}{\{\mathtt{var}\ x;\ c\} \downarrow_g \forall x : f} \qquad \frac{c_1 \downarrow_{g_1} f_1 \quad c_2 \downarrow_{g_2} f_2 \quad \mathrm{PRE}(c_2, f_2) = f_3}{\{c_1\,;c_2\} \downarrow_{g_1 \land g_2} f_1 \land f_3}$$

$$\frac{e \simeq_h f_e \qquad c \downarrow_g f}{\mathtt{if}\ (e)\ \mathtt{then}\ c \downarrow_g f_e \ \Rightarrow f} \qquad \frac{e \simeq_h f_e \qquad c_1 \downarrow_{g_1} f_1 \qquad c_2 \downarrow_{g_2} f_2}{\mathtt{if}\ (e)\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \downarrow_{g_1 \land g_2} \mathsf{if}\ f_e\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2}$$

$$\frac{\begin{array}{c} e \simeq_h f_e \qquad c : [f_c]^{xs}_{g_c,h_c} \qquad c \downarrow_{g_t} f_t \\ g \equiv \forall xs, ys, zs : f[xs/\mathsf{old}\ xs, ys/\mathsf{var}\ xs] \land f_e[ys/\mathsf{old}\ xs] \land f_c[ys/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \Rightarrow \\ g_t[ys/\mathsf{old}\ xs] \land f_t[ys/\mathsf{old}\ xs] \land \mathsf{let}\ n = t[zs/\mathsf{old}\ xs]\ \mathsf{in}\ n \in \mathbb{N} \land n < t[ys/\mathsf{old}\ xs] \end{array}}{\mathtt{while}\ (e)^{f,t}\ c \downarrow_g t \in \mathbb{N}}$$

Figure 3: The Termination Rules

where x denotes a program variable, e denotes a program expression, and a while loop is annotated by an invariant formula f and termination term t. As shown in Figures 2, 3, and 4 (where the terms $\mathsf{old}\ xs$ and $\mathsf{var}\ xs$ refer to the sets of values of the program variables xs in the pre-/post-state), we can derive for these commands the following kinds of judgments:

- $c : [f_r]^{xs}_{g,h}$ denotes the derivation of a state relation f_r from command c together with the set of program variables xs that may be modified by c. The derived relation is correct if the derived state-independent condition g holds, and if the derived state condition h holds on the pre-state of c. The rationale for g is is to capture state-independent conditions such as the correctness of loop invariants; the purpose of h is to capture statement preconditions that prevent e.g. arithmetic overflows. These side conditions have to be proved; they are separated from the transition relation f_r to make the core of the relation better understandable.
- $c \downarrow_{g_c} f_c$ denotes the derivation of a state condition (termination condition) f_c from c; the derived condition is correct, if the state-independent condition g_c holds. The purpose of this side condition

$$\frac{c : [f]^{xs}_{g,h}}{\text{PRE}(c, f_q) = \forall xs : f[xs/\mathsf{var}\ xs] \Rightarrow f_q[xs/\mathsf{old}\ xs]}$$

$$\frac{c : [f]^{xs}_{g,h}}{\text{POST}(c, f_p) = \exists xs : f_p[xs/\mathsf{old}\ xs] \wedge f[xs/\mathsf{old}\ xs, \mathsf{old}\ xs/\mathsf{var}\ xs]}$$

Figure 4: The Pre-/Postcondition Rules

is to capture that the value of a loop's termination term does not decrease forever.

- $\text{PRE}(c, f_q) = f_p$ and $\text{POST}(c, f_p) = f_q$ denote derivations that compute from a command c and a condition f_q on the post-state of c a corresponding condition f_p on the pre-state, respectively from c and pre-condition f_p the post-condition f_q. The corresponding rules in Figure 4 show that these conditions can be computed directly from the transition relation of c.

The derivations make use of additional judgments $e \simeq_{f_e} f$ and $e \simeq_{f_e} t$ which translate a boolean-valued program expression e into a logic formula f and an expression e of any other type into a term t, provided that the state in which e is evaluated satisfies the condition f_e (the rules for these judgments are omitted). Formally, the derivations satisfy the following soundness constraints.

Theorem 1 (Soundness) *For all $c \in Command, f_r, f_c, f_p, f_q, g, h \in Formula, xs \in \mathbb{P}(Variable)$, the following statements hold:*

1. *If we can derive the judgment $c : [f_r]^{xs}_{g,h}$, then we have for all $s, s' \in Store$*

$$[\![g]\!] \wedge [\![h]\!](s) \Rightarrow ([\![c]\!](s,s') \Rightarrow [\![f_r]\!](s,s') \wedge \forall x \in Variable \backslash xs : [\![x]\!](s) = [\![x]\!](s')).$$

2. *If we can (in addition to $c : [f_r]^{xs}_{g,h}$) derive the judgment $c \downarrow_{g_c} f_c$, then we have for all $s \in Store$*

$$[\![g]\!] \wedge [\![g_c]\!] \wedge [\![h]\!](s) \Rightarrow ([\![f_c]\!](s) \Rightarrow \langle\!\langle c \rangle\!\rangle(s)).$$

3. *If we can (in addition to $c : [f_r]^{xs}_{g,h}$) also derive the judgment $\text{PRE}(c, f_q) = f_p$ or the judgment $\text{POST}(c, f_p) = f_q$, then we have for all $s, s' \in Store$*

$$[\![g]\!] \wedge [\![h]\!](s) \Rightarrow ([\![f_p]\!](s) \wedge [\![f_r]\!](s,s') \Rightarrow [\![f_q]\!](s')).$$

The semantics $[\![f]\!](s,s')$ of a transition relation f is determined over a pair of states s, s' (and a logic environment, which is omitted for clarity); the semantics of state condition g is defined as $[\![g]\!](s) \Leftrightarrow \forall s' : [\![g]\!](s,s')$ and the semantics of a state independent-condition h is defined as $[\![h]\!] \Leftrightarrow \forall s, s' : [\![h]\!](s,s')$.

As an example, take the following method *fac* computing the factorial of a natural number n (the specification term `VALUE@NEXT` denotes the return value of *fac*):

```
public static int fac(int n) /*@
  requires OLD n >= 0 AND factorial(OLD n) <= Base.MAX_INT;
  ensures VALUE@NEXT = factorial(OLD n); @*/ {
  int i=1; int p=1;
  while (i <= n) /*@
    invariant OLD n >= 0 AND factorial(OLD n) <= Base.MAX_INT
          AND 1 <= VAR i AND VAR i <= OLD n+1 AND VAR p = factorial(VAR i-1);
    decreases OLD n - OLD i + 1; @*/ {
    p = p*i; i = i+1;
  }
  return p;
}
```

Based on the calculus above, the RISC Program Explorer translates the while loop to the following formulas (and also generates tasks for the verification of the various side conditions):

Effects

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

Transition Relation

$\text{var } i = (\text{old } n+1) \wedge \text{old } n \geq 0 \wedge \text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}} \wedge 1 \leq \text{var } i$
$\wedge$
$\text{var } p = \text{factorial}(\text{var } i-1)$

Termination Condition

$(\text{old } n - \text{old } i+1) \geq 0$

Here the core of the transition relation is the formula $\mathsf{var}\ i = \mathsf{old}\ n+1 \wedge \mathsf{var}\ p = \mathit{factorial}(\mathsf{var}\ i-1)$ while the termination condition is $\mathsf{old}\ n - \mathsf{old}\ i+1 \geq 0$. Based on this translation, the body of the method *fac* is translated to

Transition Relation

$(\exists i \in \text{Base.int}: i = (\text{old } n+1) \wedge 1 \leq i \wedge \text{value@next} = \text{factorial}(i-1)) \wedge \text{old } n \geq 0$
$\wedge$
$\text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}} \wedge \text{returns@next}$

Termination Condition

$\forall \text{now} \in \text{S}(\text{Base.int}): \text{executes@now} \wedge \text{executes@now} \Rightarrow \text{old } n \geq 0$

Here the core of the transition relation is $\exists i : i = \mathsf{old}\ n+1 \wedge \mathit{value}\mathtt{@next} = \mathit{factorial}(i-1)$ which can be further simplified to $f_r :\Leftrightarrow \mathit{value}\mathtt{@next} = \mathit{factorial}(\mathsf{old}\ n)$; the termination condition can be further simplified to $f_c :\Leftrightarrow \mathsf{old}\ n \geq 0$ (work is going on to improve the automatic simplification). Both f_r and f_c represent the semantic essence of *fac* from which the correctness of the method according to its specification is quite self-evident even before the formal proof is started. More realistic examples seem to indicate that from the construction and simplification of the semantic essence also the further verifications become substantially clearer and perhaps even technically simpler.

References

[1] Bernhard Beckert, Reiner Hähnle & Peter H. Schmitt, editors (2007): *Verification of Object-Oriented Software: The KeY Approach*. *Lecture Notes in Computer Science* 4334, Springer-Verlag. `http://www.springer.com/computer/ai/book/978-3-540-68977-5`.

[2] Leslie Lamport (2002): *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. `http://research.microsoft.com/users/lamport/tla/book.html`.

[3] S. Owre, J. M. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System*. In Deepak Kapur, editor: *11th International Conference on Automated Deduction (CADE)*, *Lecture Notes in Artificial Intelligence* 607, Springer, Saratoga, NY, June 14–18, pp. 748–752. `http://www.csl.sri.com/papers/cade92-pvs`.

[4] David A. Schmidt (1986): *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Boston, MA. `http://people.cis.ksu.edu/~schmidt/text/densem.html`.

[5] Wolfgang Schreiner (2008): *The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom*. *Formal Aspects of Computing* doi:10.1007/s00165-008-0069-4.