

Program Verification with the RISC ProofNavigator

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

Abstract

This paper describes the use of the RISC ProofNavigator, an interactive proving assistant for the area of program verification. This assistant has been developed with a focus on simplicity and ease of use; it is intended to be suitable for educational scenarios as well as for realistic applications.

Keywords: Interactive Proving Assistants, Computer-Aided Verification

1. INTRODUCTION

While formal methods become more and more visible in computer science curricula, students are frequently taught the basic concepts (verification calculi, specification languages) without actually gaining major practical experience with formal correctness proofs. Lecturers tend to shy away from performing such proofs in lecture halls and laboratories because on the one hand paper-and-pencil proofs are complex, error-prone, and hardly ever convincing and on the other hand the students' learning curve of existing proving assistants is considered too steep for the available budget of teaching time. Consequently, most software engineers enter the market without ever having attempted formal proofs; this perpetuates the general opinion that the only value of formal methods (if any) is in areas where fully automatic tools hide the underlying theory (e.g. model checking of finite state systems), an opinion that is not so dominant in any other engineering discipline. That the act of formal reasoning on certain aspects of a system may actually help to gain *insight* into the system is completely neglected.

While a variety of powerful theorem provers and interactive proving assistants have become available [7], it is indeed true that many of them are difficult to learn or inconvenient to use (but more and more effort is also put on the user interface aspects of provers [2, 1]). Based on a number of use cases derived from the area of program verification, the author evaluated from 2004 to 2005 a couple of prominent systems. While we achieved quite good results with PVS [4], we generally encountered various problems and nuisances, especially with the navigation within proofs, the presentation of proof states, the treatment of arithmetic, and the general interaction of the user with the systems; we frequently found that the elaboration of proofs was more difficult than we considered necessary. Without any doubt, some of these problems were caused by our own inabilities and could have been overcome by more training and experience but this is exactly the hurdle for the more wide-spread application of this kind of software.

From these experiments, we also drew a couple of important conclusions for the pragmatics of using a proving assistant in program verification:

- Convenient navigation in proof trees is essential; the user gets easily lost in large proofs.
- The aggressive simplification of proof state descriptions and their comfortable presentation is important, since the user quickly loses intuition about the interpretation of a proof situation.
- Decent automation in dealing with arithmetic is important; a subtype relationship between integers and reals simplifies some proofs considerably.

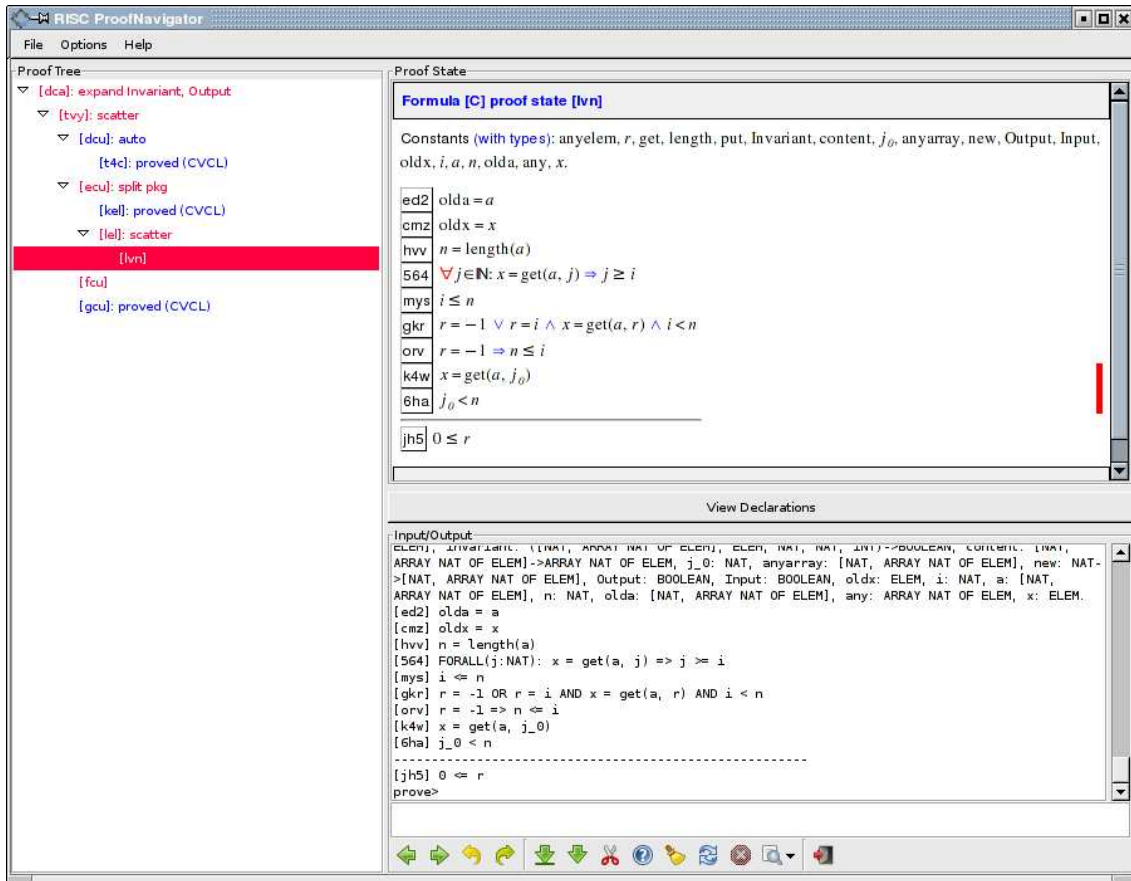


FIGURE 1: The RISC ProofNavigator in Action

- Automatic search for proofs based on elaborate strategies is rarely of much help; typically it is the combination of semi-automatic proof decomposition, critical hints given by the user, and the application of decision procedures for ground theories that shows practical success.

Based on above investigations we decided to write a new proof assistant to meet above criteria (with various features copied from PVS and adapted to our taste). This task became possible with reasonable effort by making use of existing software that decides about the satisfiability of formulas over certain combinations of ground theories. During the last couple of years, various tools for solving this *SMT (satisfiability modulo theories)* problem have emerged [6]. Around one such tool, the *Cooperating Validity Checker Lite (CVCL)* [3], we developed the *RISC ProofNavigator* [5], a proving assistant which shall be suitable as well in educational as in real application scenarios. The software is implemented in Java, runs on GNU/Linux computers with x86-compatible processors, and is freely available as open source. The remainder of the paper focuses on the practical use of this software; its specification language and underlying logic are described in the software documentation.





2. THE SOFTWARE






When the RISC ProofNavigator is started, a window pops up displaying three areas (see Figure 1 which depicts the software in the course of the proof presented in the following section):


Proof Tree This area illustrates the skeleton structure of the proof which is currently investigated. It mainly serves for easy navigation: a click on a tree node displays the corresponding proof state; a double click switches to that state in order to apply a proof command.

Proof State This area initially displays the declarations entered by the user in a pretty-printed form which closely resembles the usual mathematical notation (while the output window below shows a corresponding plain text notation). In proving mode, this area displays in Gentzen-style the proof state currently investigated by the user as two sequences of formulas, the “assumptions” and the “goals”, separated by a horizontal line. The obligation is to prove that the conjunction of the assumptions implies the disjunction of the goals.

Input/Output This area consists of an input field where the user may type in declarations and commands, an output field where the effect of the user input is shown as plain text, and a row of buttons that serve two purposes:

Proof Navigation The buttons  (“previous open state”),  (“next open state”),  (“undo command”),  (“redo command”) allow circling through the list of open proof states, undoing the effect of a proof command (thus discarding a subtree) and restoring a discarded proof tree again.

Proof Control The other buttons give the user access to the most important high-level strategies for decomposing a proof and/or closing a proof state. For instance, the button  (“scatter state”) recursively applies logical decomposition rules such as “ \forall -introduction”, “ \wedge -introduction”, etc. to the current proof state and to all generated child states and also attempts to close the generated states by the application of decision procedures. Less aggressive decomposition strategies are applied by the buttons  (“decompose state”) and  (“split state”); the buttons  (“close state by automatic instantiation of formulas”) and  (“apply formula instantiation also to sibling states”) apply heuristics for the instantiations of universally quantified assumptions respectively existentially quantified goals.

By pressing the button , a menu of all available proof commands is displayed. Finally, by moving the mouse cursors over the label attached to a formula, a menu pops up that displays commands that may be applied in the current proof state and relate to that formula (e.g. to use a disjunctive assumption to split a proof state into multiple child states).

The user interface was designed with a couple of specific goals in mind, such as to maximize survey on the overall proof situation, to minimize the number of options, and in general to minimize the efforts to interact with the software and to elaborate a proof. We believe that after a short learning period the interface becomes very transparent such that the user can concentrate on the mathematical aspects of a proof rather than on handling the software. By convenient navigation and undo/redo mechanisms one can quickly browse through a proof and experiment with different proving strategies.

3. A VERIFICATION EXAMPLE

We demonstrate the use of the software by (a part of) the verification of the following Hoare triple; this triple represents the core of a program which searches in an array a for the smallest index r at which an element x is stored:

```

{olda = a ∧ oldx = x ∧ n = |a| ∧ i = 0 ∧ r = -1}
while i < n ∧ r = -1 do
  if a[i] = x
    then r := i
    else i := i + 1
{a = olda ∧ x = oldx ∧
((r = -1 ∧ ∀i : 0 ≤ i < |a| ⇒ a[i] ≠ x) ∨ (0 ≤ r < |a| ∧ a[r] = x ∧ ∀i : 0 ≤ i < r ⇒ a[i] ≠ x))}

```

By the rules of the Hoare calculus, the verification of this triple is reduced to the proof of a couple of verification conditions, one of which is

$$Invariant \wedge \neg(i < n \wedge r = -1) \Rightarrow Output$$

Here *Output* represents the postcondition of the above triple and *Invariant* is a suitable loop invariant. We are going to demonstrate this condition's proof which will ultimately have the following tree structure:

```
[dca]: expand Invariant, Output in zfg
      [tvj]: scatter
            [dcu]: auto
                  [t4c]: proved (CVCL)
            [ecu]: split pkg
                  [kel]: proved (CVCL)
            [lej]: scatter
                  [lvn]: auto
                        [lap]: proved (CVCL)
            [fcu]: auto
                  [blt]: proved (CVCL)
            [gcu]: proved (CVCL)
```

This tree has seven inner nodes representing invocations of the commands `expand`, `scatter`, `auto`, and `split` by the user; it has five leaf nodes which were automatically closed by the underlying decision procedure (CVCL).

The root state [dca] has goal [zfg] with occurrences of the predicates *Invariant* and *Output*.

Formula [C] proof state [dca] : expand Invariant, Output in zfg

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

[zfg] Invariant(*a*, *x*, *i*, *n*, *r*) \Rightarrow Output $\vee i < n \wedge r = -1$

Children: [tvj]


We use the command `expand Invariant, Output in zfg` to replace these predicates by their definitions, which results in the following state:

Formula [C] proof state [tvj] : scatter

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

```
aqc  olda = a  $\wedge$  oldx = x  $\wedge$  n = length(a)  $\wedge$  ( $\forall j \in \mathbb{N}: x = \text{get}(a, j) \Rightarrow j \geq i$ )
 $\Rightarrow$ 
      n < i  $\vee r \neq -1 \wedge (x = \text{get}(a, r) \wedge r = i \Rightarrow n \leq i) \vee i < n \wedge r = -1$ 
 $\vee$ 
      olda = a
 $\wedge$ 
      ( r = -1  $\wedge$  ( $\forall j \in \mathbb{N}: x = \text{get}(a, j) \Rightarrow j \geq \text{length}(a)$ )
 $\vee$ 
        0  $\leq r \wedge x = \text{get}(a, r) \wedge r < \text{length}(a)$ 
 $\wedge$ 
        ( $\forall j \in \mathbb{N}: x = \text{get}(a, j) \Rightarrow j \geq r$ )
```

Parent: [dca] Children: [dcu] [ecu] [fcu] [gcu]


We do not bother to investigate the structure of this state further but immediately press the “Scatter” button  which generates four children states of which one is closed automatically. Of the three remaining states [dcu], [ecu], and [fcu], the first one is as follows:

Formula [C] proof state [dcu] : auto

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

```
ed2  olda = a
cmz  oldx = x
hvv  n = length(a)
564   $\forall j \in \mathbb{N}: x = \text{get}(a, j) \Rightarrow j \geq i$ 
mys   $i \leq n$ 
x2w   $r = -1$ 
cpb   $n \leq i$ 
k4w   $x = \text{get}(a, j_0)$ 
6ha   $j_0 < n$ 
f5e   $x = \text{get}(a, -1)$ 
```

Parent: [tvv] Children: [t4c]

The state has an universally quantified assumption [564]; we need to use a proper instantiation of this formula. Since also the other two open states may have similar structures, we press  which heuristically instantiates universally quantified assumptions in all these states. Indeed both [dcu] and [fcu] are closed automatically such that we only need to investigate state [ecu] further:

Formula [C] proof state [ecu] : split pkg

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

```
ed2  olda = a
cmz  oldx = x
hvv  n = 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   $r = -1 \Rightarrow n \leq i$ 
pkg   $r = -1 \Rightarrow (\exists j \in \mathbb{N}: x = \text{get}(a, j) \wedge j < \text{length}(a))$ 
jh5   $0 \leq r$ 
```

Parent: [tvv] Children: [kel] [lel]


This state has three assumptions that start with the formula $r = -1$ (which denotes “element not found” in the program). Our further reasoning depends on which of the two possibilities $r = -1$ or $r \neq -1$ is true. From the menu behind the formula label [pkg] we can select the command `split` which “splits” the current state into several children each of which receives as an additional assumption one of the components of the disjunctive formula. From the resulting two child states one is automatically closed while the other with label [lel] still requires our attention:

Formula [C] proof state [lel] : scatter

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

```
ed2  olda = a
cmz  oldx = x
hvv  n = 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   $r = -1 \Rightarrow n \leq i$ 
1bb   $\exists j \in \mathbb{N}: x = \text{get}(a, j) \wedge j < \text{length}(a)$ 
jh5   $0 \leq r$ 
```

Parent: [ecu] Children: [lvn]


This state has an existential assumption [1bb]. To get rid of the quantifier, we press the “Scatter” button  and get the state [lvn]:

Formula [C] proof state [lvn] : auto

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

ed2	olda = a
cmz	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	$j_0 < n$
<hr/>	
jh5	$0 \leq r$

Parent: [lel] Children: [lap]

This state contains a universally quantified assumption [564]; we try automatic instantiation of this formula with the “Auto” button . Indeed, a proof state [lap] is generated which is automatically closed such that the proof is completed.

4. CONCLUSIONS

We believe that the repertoire of every decent computer scientist should (based on a sound education in mathematical logic) also comprise the use of a proving assistant and that practical experience with some tool of this kind should be part of every computer science curriculum. The RISC ProofNavigator is an attempt to help to achieve this goal. We have actually not yet applied the software in a classroom but we plan to do so in 2007 in a regular course on formal methods (where we currently use PVS); another lecturer will investigate its use in a course on formal reasoning (which currently does not apply any tools at all). Furthermore, the RISC ProofNavigator has been successfully applied to verifications that were already so complex that they were very difficult to manage with some other tools of this kind; the software should therefore be also suitable for non-classroom scenarios. Indeed our long-term goal is the development of an integrated program reasoning environment which includes the RISC ProofNavigator as a core component.

REFERENCES

- [1] J.-R. Abrial and D. Cansell. Click’n Prove: Interactive Proofs within Set Theory. In D. A. Basin and B. Wolff, editors, *TPHOLs 2003*, volume 2758 of *LNCS*, pages 1–24. Springer, 2003.
- [2] D. Aspinall et al., editors. *User Interfaces for Theorem Provers*, Satellite Workshop of ETAPS 2005, Edinburgh, UK, April 9, 2005. <http://homepages.inf.ed.ac.uk/da/uitp05>.
- [3] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13–17, 2004*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.
- [4] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 14–18, 1992. Springer.
- [5] The RISC ProofNavigator, 2006. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, <http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator>.
- [6] SMT-LIB — The Satisfiability Modulo Theories Library, 2006. University of Iowa, Iowa City, IA, <http://combination.cs.uiowa.edu/smtlib>.
- [7] F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer, Berlin, 2006.