

The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom

Wolfgang Schreiner

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

Abstract. This paper gives an overview on the RISC ProofNavigator, an interactive proving assistant for the area of program verification. The assistant combines the user-guided top-down decomposition of proofs with the automatic simplification and closing of proof states by an external satisfiability solver. The software exhibits a modern graphical user interface which has been developed with a focus on simplicity in order to make the software suitable for educational scenarios. Nevertheless, some use cases of a certain level of complexity demonstrate that it may be also appropriate for various realistic applications.

Keywords: Interactive Proving Assistants, Computer-Aided Verification, Teaching Formal Methods.

1. Introduction

Formal methods [HR04] become more and more visible in computer science curricula, but typically for the purpose of modeling hardware and software systems rather than for actually reasoning about them. Lecturers tend to shy away from performing formal proofs of system properties in the classroom such that most software engineers enter the market without ever having attempted such 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. the model checking of finite state systems). That the act of formal reasoning on certain aspects of a system may actually help to gain *insight* into the system is completely neglected.

One major reason that the aspect of proving is underrated in computer science curricula is that manual system proofs are complex, error-prone, and hardly ever convincing. However, in the last two decades, a variety of powerful automatic theorem provers and interactive proving assistants have become available many of which can be also applied to the area of program and system verification [Wie06]. For instance, the prototype verification system *PVS* [ORS92] is an integrated specification and verification environment

based on higher order logic. *Isabelle* [NPW05] is a generic proving environment into which various logics can be embedded; the most widely used variant is Isabelle/HOL which uses higher order logic. *Coq* [BC04] is based on the calculus of constructions; from proofs in *Coq*, programs can be extracted. *ACL2* [KMM00] is the latest incarnation of the Boyer-Moore family of provers based on a first-order logic of total recursive functions. *Theorema* [BCJ⁺06] is a system that generates human-readable automatic proofs by a combination of a higher-order predicate logic prover and a collection of special provers. Also CASE tools have been augmented by capabilities for formal specification and verification: the *KeY* tool [BHS07] for the development of JavaCard programs includes a proving assistant for first-order dynamic logic; *Perfect Developer* is a commercial object-oriented development tool that includes a fully automatic theorem prover [CMM05].

While thus a variety of tools for supporting reasoning are around, the available budget of teaching time is frequently considered to small to introduce students to their practical use. Indeed, while more and more effort is also put on the user interface aspects of such systems [A⁺05, AC03], it has to be admitted that many of them are difficult to learn and/or inconvenient to use, which makes them less suitable for classroom scenarios [Fei05]. We (the author) had the same experience when we evaluated from 2004 to 2005 a couple of prominent proving assistants by a number of use cases derived from the area of program verification. While we achieved quite good results with PVS, 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 drew a couple of important conclusions for the pragmatics of using a proving assistant in program verification (in classroom and elsewhere):

- The presentation of a proof as a tree of all proving states (as opposed to the presentation as a flat list of only the open proving tasks) is crucial to keep track of the overall situation; correspondingly, facilities for convenient navigation in proof trees are essential.
- The aggressive simplification of proof state descriptions and their comfortable presentation (in particular an adequate layout of large formulas) is important; the user quickly loses intuition about the interpretation of a proof situation.
- Decent automation in dealing with arithmetic is important; furthermore, a subtype relationship between integers and reals (rather than explicit conversion functions between the two domains) simplifies some proofs considerably.
- 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 these considerations we decided to write a proving assistant to meet above criteria (with various features copied from PVS and adapted to our taste). We reckoned that this task would become 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 [SMT06]. Around one such tool, the *Cooperating Validity Checker Lite (CVCL)* [BB04], we finally developed the *RISC ProofNavigator* [Sch06b], 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 [RIS06].

The remainder of this paper is structured as follows: in Section 2, we describe the system's user interface and the corresponding design considerations; Section 3 sketches the implementation of the system; Section 4 presents the specification language and proof logic; Section 5 walks through an example verification; Section 6 surveys the application of the software to various use cases; Section 7 concludes and gives an outlook on further development.

This paper is a substantial revision and extension of [Sch06a] from which the verification example presented in Section 5 is taken.

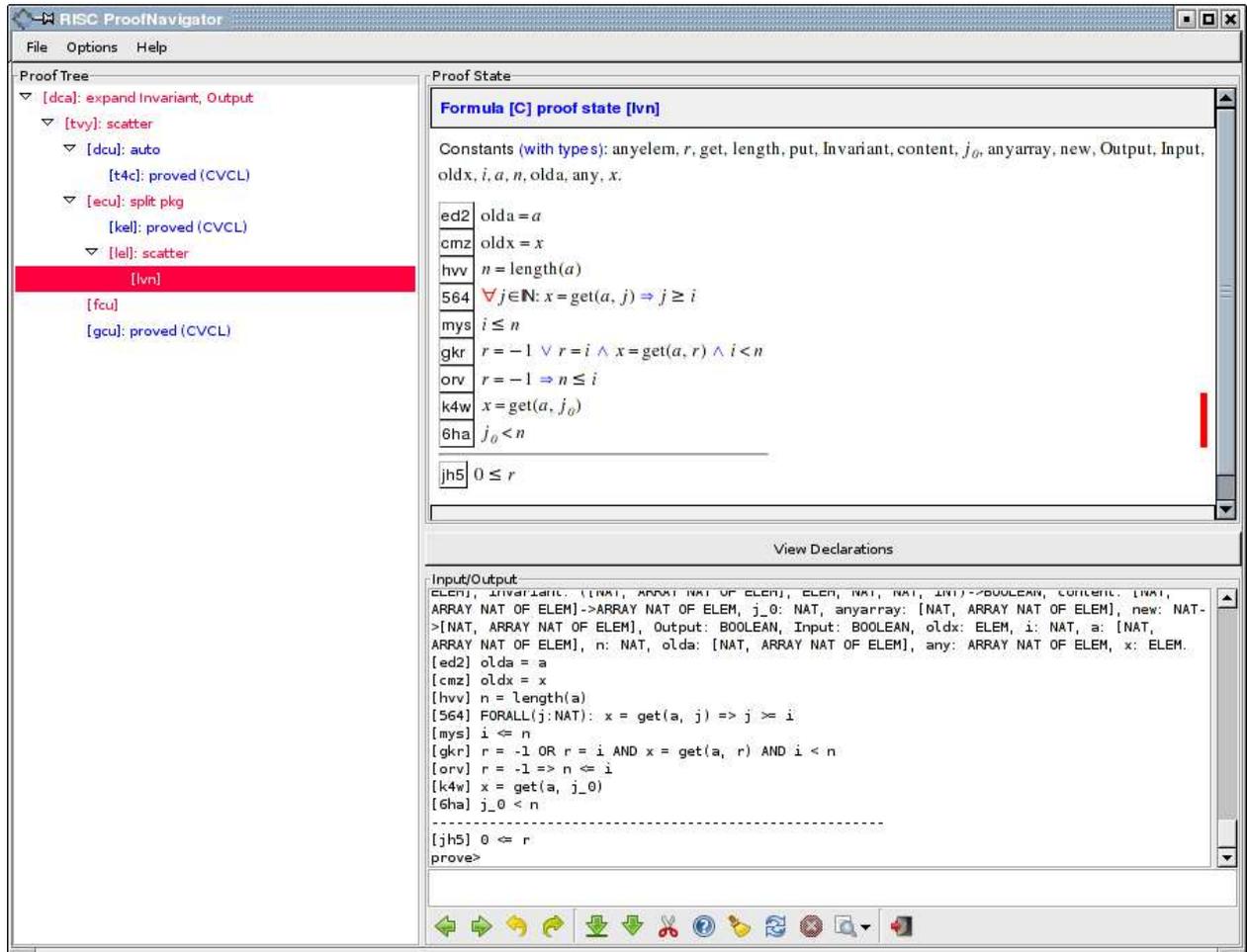


Fig. 1. The RISC ProofNavigator in Action

2. The User Interface

When the RISC ProofNavigator is started, a window pops up that displays three major areas (see Figure 1):

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 Initially this area displays the declarations entered by the user (respectively read from a file) in a pretty-printed form which closely resembles the usual mathematical notation (the output window below shows a corresponding plain text notation). When a proof is started, this area displays the proof state which is currently investigated.

Input/Output This area consists of an input field where the user may type in declarations and commands and an output field where the effect of the user input is shown as plain text. The user may directly type in a command or select it from a menu as a template with gaps for the individual arguments: by pressing the tabulator button, the user jumps from one gap to the next to enter the necessary data.

Furthermore, there is a row of buttons that serve two purposes:

Proof Navigation The buttons  (“previous open state”),  (“next open state”),  (“undo com-

mand”),  (“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 the underlying satisfiability solver. 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 button icons for proof control were chosen to visually resemble their effects, for instance the icon  (“decompose state”) is expanded to expand the proof tree “downwards” (towards the leaves) in a limited fashion, while  (“scatter state”) expands the proof tree until no further automatic expansion is possible. Correspondingly,  (“close state by automatic instantiation of formulas”) is intended to “refresh” the view of a proof state and  (“apply formula instantiation also to sibling states”) “cleans up” a sequence of proof states. All buttons are equipped with explanatory popup menus such that their meaning should become quickly apparent also to new users.

The user interface was deliberately designed with various goals in mind:

Maximize Survey The user should easily keep a general view on proofs with many states; she should also easily keep control on proof states with large numbers of potentially large formulas. These purposes are served by the proof tree area and by the pretty-printing of formulas with appropriate indentations. Furthermore, before presentation every proof state is forwarded to a decision/simplification procedure; only if it cannot be automatically closed, it is displayed after simplification by logical and arithmetic transformations.

Minimize Options The number of commands is deliberately kept as small as possible in order to minimize confusion and simplify the learning process (in total there are about thirty commands, of which only twenty are actually proving commands; in typical proofs, less than ten commands need to be used). In particular, there are only commands to control the predicate-logic structure of a proof; reasoning on the level of terms and arithmetic is completely left to the underlying satisfiability solver (which decides linear integer arithmetic; non-linear arithmetic properties can be introduced as lemmas and subsequently verified by induction).

Minimize Efforts The most important commands can be triggered by buttons or by menu entries attached to formula labels. The keyboard only needs to be used in order to enter terms for specific instantiations of universal assumptions or existential goals (but even here typing effort is minimized by the selection of command “templates” from menus).

Formulas are presented in a visually appealing mathematical form (in the proof state area) as well as in a linear textual form (in the output area); the later form on the one hand uniquely exhibits the logical structure of a formula (which is useful if uncertainties arise) and on the other hand is convenient for copying&pasting formulas and terms into the input area (e.g. for the instantiation of a quantified formula). Every formula is tagged with a label of the form $[xyz]$ where xyz are three characters derived from hashing the text of the formula (after a canonical renaming of the variables bound in the formula); this allows to refer to formulas independent of their positions in a proof state, which makes proofs whose commands refer to specific formulas more robust against changes in definitions (compared to systems such as PVS or Isabelle where commands depend on the ordering of formulas and even minor changes in definitions subsequently break proofs).

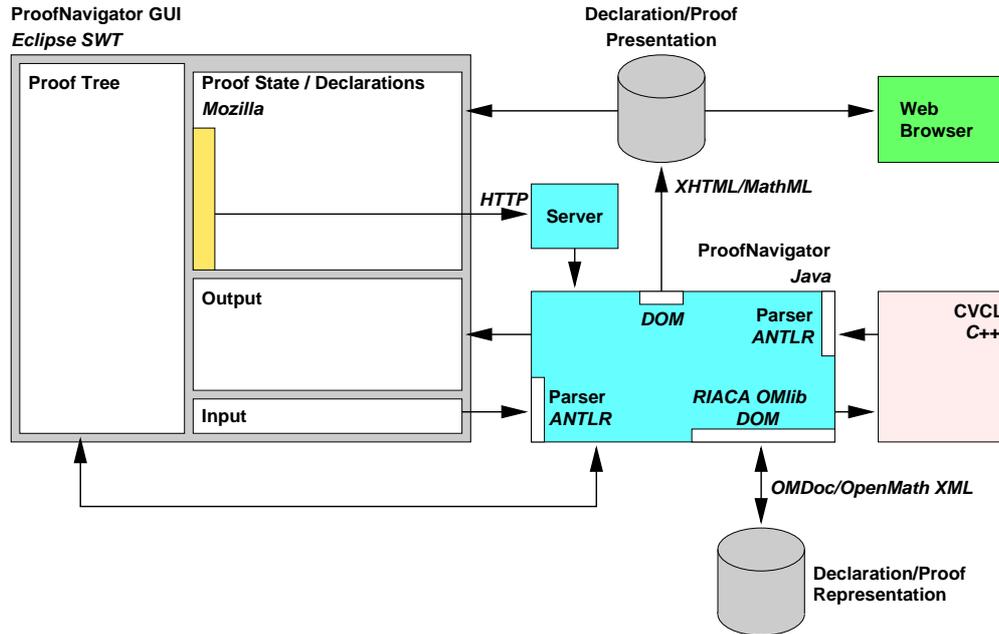


Fig. 2. The Software Architecture of the RISC ProofNavigator

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. The Software Architecture

The RISC ProofNavigator has been entirely implemented with free software. It consists of the following components (see Figure 2):

The Graphical User Interface The GUI is written in Java with the use of the Eclipse Standard Widget Toolkit (SWT) and embeds the Mozilla browser as a component for the presentation of declarations and proofs. User interactions in the browser component (selections of menu entries) are forwarded as HTTP requests to a thread that acts as an HTTP server and triggers the corresponding activity in the ProofNavigator kernel.

The ProofNavigator Kernel The kernel of the system is written in Java; it includes parsers generated with ANTLR for processing the Proof Navigator specification language and the output of the CVCL software. Using the Java Document Object Model (DOM) interface, it generates external representations of declarations and proofs that remain persistent across sessions.

The Cooperating Validity Checker (CVCL) CVCL is written in C++; instances of the software are incorporated as external processes that interact with the kernel via their standard input/output interfaces. One instance is created for simplifying user declarations and one instance is created for the simplification of every proof state.

For the external representation of data, open XML-based standards are used:

Presentation The kernel generates XHTML documents which embed MathML expressions for the presentation of declarations and proofs by the GUI's browser component. These documents are appropriately hyper-linked and pertained across sessions such that they can be used for presentation in the Web by standard-compliant browsers (such as the browsers of the Mozilla suite).

Session Store Declarations are stored for use in later sessions as (compressed) OMDoc documents [Koh06]

embedding the XML representations of OpenMath objects [Boe04] denoting types, functions, and predicates; these documents may thus in principle be processed by other OMDoc-compliant tools. Likewise, proof trees are stored for later replay in an XML representation where for every node of the tree (only) the proof command issued in the corresponding proof situation is stored; the representation is ad-hoc because OMDoc does not support a notion of proofs suitable for our purpose. For details on the representation formats, see the manual [Sch06b].

The session store maintains the dependencies among declarations and proofs as well as the status of every proof as “open” or “completed”. If a new session is started, the dependencies are used to re-assess the status of every proof: if the proof only depends on declarations whose dependencies have not changed, its status is tagged as “trusted”; if however, some dependency (e.g. the declaration of a formula on which the proof depends) has changed, its status is tagged as “untrusted” indicating that the stored proof should be replayed to re-establish its trust level. The explicit maintenance of proofs is very important, because the proof development process typically requires continuous modifications of definitions and thus more than one proving session; nevertheless it is (with the major exception of PVS) not supported by most other assistants.

If an untrusted proof is replayed from its representation in the session store, the kernel traverses the proof tree and executes the proving commands indicated by the nodes of the tree. If a proving command fails, the proof replay suspends in the denoted proof situation and prompts the user for help. The user has now the chance to abort the proof situation (i.e. the original subtree rooted in that node is discarded and the replay continues with the other nodes) or providing a substitution command with which the replay attempts to continue the proof (as if no error had occurred). This possibility is rather important in those cases where the replay has only failed because of some minor change (such as the label of a formula referenced in a proof command) but the proof remains structurally intact; while in PVS the user has to construct the corresponding proof subtree again from scratch, it can with little effort be recovered in the RISC ProofNavigator.

4. The Specification Language and Proof Logic

The RISC ProofNavigator uses a language in the style of PVS respectively CVCL for the specification of types, constants, functions, predicates, axioms, and ultimately formulas to be proved. The language is based on a higher-order logic and includes boolean values, integer numbers, real numbers, tuples, functions, arrays, and records as basic datatypes respectively type constructors. The language also supports subtypes, i.e., types that are defined as those elements of some other type that satisfy some (user-defined) condition. Type checking is based on a combination of static type checking for the basic types and the verification of automatically generated type checking conditions for the subtypes.

The full language is described in the manual [Sch06b]. As an illustrative example, we define a datatype “finite-length arrays” with corresponding constructor and selector operations. Every element of this datatype is a tuple of a function that maps the natural numbers to array values (based on the internal notion of arrays as unary functions) and a natural number that denotes the length of the array (and thus constrains the range of indices to which the function may be applied). The corresponding type declarations are

```
INDEX: TYPE = NAT;
ELEM:  TYPE;
ARR:   TYPE = [INDEX, ARRAY INDEX OF ELEM];
```

We then introduce various auxiliary constants whose values remain undefined and that serve in the later declarations as error signals:

```
any:      ARRAY INDEX OF ELEM;
anyelem: ELEM;
anyarray: ARR;
```

We also define the following auxiliary function constant:

```
content: ARR -> (ARRAY INDEX OF ELEM) = LAMBDA(a:ARR): a.1;
```

The value of this constant is the function that, given an argument a of type `ARR`, returns its second component $a.1$, i.e. the content of the array (the notation $a.i$ denotes component i of tuple a ; components are numbered starting with 0). With the help of these auxiliary notions, we define our core functions on arrays:

```

length: ARR -> INDEX = LAMBDA(a:ARR): a.0;

new: INDEX -> ARR = LAMBDA(n:INDEX): (n, any);

put: (ARR, INDEX, ELEM) -> ARR =
  LAMBDA(a:ARR, i:INDEX, e:ELEM):
    IF i < length(a)
      THEN (length(a), content(a) WITH [i]:=e)
      ELSE anyarray
    ENDIF;

get: (ARR, INDEX) -> ELEM =
  LAMBDA(a:ARR, i:INDEX):
    IF i < length(a) THEN content(a)[i] ELSE anyelem ENDIF;

```

The adequacy of these definitions is stated by the declarations of the following formulas which say that the functions satisfy the classical properties required from arrays:

```

length1: FORMULA
  FORALL(n:INDEX): length(new(n)) = n;

length2: FORMULA
  FORALL(a:ARR, i:INDEX, e:ELEM):
    i < length(a) => length(put(a, i, e)) = length(a);

get1: FORMULA
  FORALL(a:ARR, i:INDEX, e:ELEM):
    i < length(a) => get(put(a, i, e), i) = e;

get2: FORMULA
  FORALL(a:ARR, i, j:INDEX, e:ELEM):
    i < length(a) AND j < length(a) AND i /= j =>
      get(put(a, i, e), j) = get(a, j);

```

These declarations are pretty-printed by the RISC ProofNavigator as shown in Figure 3. To start the proof of e.g. formula `get2`, we may either enter the command `prove get2` or simply select this command from the menu hidden behind the label of this formula. This yields the initial proof state

Formula [get2] proof state [adu] : expand length, get, put, content	
Constants (with types): anyelem, get, length, put, content, anyarray, new, any.	
wv6	$\forall a \in \text{ARR}, i \in \text{INDEX}, j \in \text{INDEX}, e \in \text{ELEM}:$ $i < \text{length}(a) \wedge j < \text{length}(a) \Rightarrow i = j \vee \text{get}(\text{put}(a, i, e), j) = \text{get}(a, j)$
Children: [c3b]	

The presentation of the proof state consists of the value constants visible in the state and of two sequences of labelled formulas separated by a horizontal line: the *assumptions* and the *goals* (initially there are no assumptions and a single goal `[vvg]` which represents the formula `get2` to be proved). The logical interpretation of this presentation is that of a Gentzen-style sequent [Gal86]

$$A_1, \dots, A_n \vdash B_1, \dots, B_m$$

with assumptions A_1, \dots, A_n and goals B_1, \dots, B_m . This sequent can be viewed as the task to prove the validity of the formula

$$A_1 \wedge \dots \wedge A_n \Rightarrow B_1 \vee \dots \vee B_m$$

i.e. that the conjunction of the assumptions logically implies the disjunction of the goals. This is in particular achieved, if it can shown that some of the assumptions is false or some of the goals is true.

	INDEX \in type = \mathbb{N}
	ELEM \in type
	ARR \in type = [INDEX, array INDEX of ELEM]
	any \in array INDEX of ELEM
	anyelem \in ELEM
	anyarray \in ARR
	content \in ARR \rightarrow array INDEX of ELEM = $\lambda a \in$ ARR: $a.1$
	length \in ARR \rightarrow INDEX = $\lambda a \in$ ARR: $a.0$
	new \in INDEX \rightarrow ARR = $\lambda n \in$ INDEX: (n, any)
	put \in (ARR, INDEX, ELEM) \rightarrow ARR =
	$\lambda a \in$ ARR, $i \in$ INDEX, $e \in$ ELEM:
	if $i < \text{length}(a)$ then
	$(\text{length}(a), \text{content}(a) \text{ with } [i] := e)$ else
	anyarray endif
	get \in (ARR, INDEX) \rightarrow ELEM =
	$\lambda a \in$ ARR, $i \in$ INDEX: if $i < \text{length}(a)$ then $\text{content}(a)[i]$ else anyelem endif
	length1 $\equiv \forall n \in$ INDEX: $\text{length}(\text{new}(n)) = n$
	length2 \equiv
	$\forall a \in$ ARR, $i \in$ INDEX, $e \in$ ELEM: $i < \text{length}(a) \Rightarrow \text{length}(\text{put}(a, i, e)) = \text{length}(a)$
	get1 $\equiv \forall a \in$ ARR, $i \in$ INDEX, $e \in$ ELEM: $i < \text{length}(a) \Rightarrow \text{get}(\text{put}(a, i, e), i) = e$
	get2 \equiv
	$\forall a \in$ ARR, $i \in$ INDEX, $j \in$ INDEX, $e \in$ ELEM:
	$i < \text{length}(a) \wedge j < \text{length}(a) \wedge i \neq j \Rightarrow \text{get}(\text{put}(a, i, e), j) = \text{get}(a, j)$

Fig. 3. Array Declarations

Proofs are constructed by the application of a rule to the current proof state such that the system

- either recognizes the current state to denote a valid formula (in the sense described above),
- or a set of child states are generated such that the validity of the formulas denoted by these states implies the validity of the formula denoted by the current state.

By recursive rule application, thus a proof tree is constructed; the proof is completed, if all leaves of the tree are recognized to denote valid formulas.

Since $A_1 \wedge A_2 \Rightarrow B_1 \vee B_2$ is logically equivalent to $A_1 \wedge A_2 \wedge \neg B_1 \Rightarrow B_2$, by negation every formula can be switched from the list of goals to the list of assumptions and vice versa. PVS applies this property to avoid in proof state presentations a negation symbol at the outermost level of every formula by automatically presenting it as an assumption or as a goal. We found this automatism quite disturbing, because the user typically focuses on a particular formula as the goal to be proved and considers all others (negated or not) as corresponding assumptions; likewise, in a proof by contradiction, all formulas are considered as assumptions whose conjunction is shown to be inconsistent. Furthermore, to have more than one goal formula hardly corresponds to any user intuition at all.

The proving rules of the RISC ProofNavigator are therefore designed to generate a single goal formula (multiple goal formulas are still supported but only arise from the explicit manipulation of the proof state by the user, see below). Moreover, some rules treat the goal formula specially, e.g. the command `scatter` restricts the branching of the proof tree by applying those rules that may generate several child states only to the goal formula, not to any of the assumptions. The user may always choose to change her focus by explicitly “flipping” formulas between the list of assumptions and the list of goals and thus control the further proof elaboration. The automatic proof state simplification always draws negations into formulas such that e.g. a universal assumption becomes an existential goal and vice versa (while such formulas are logically equivalent, they actually represent rather different user intuitions).

Returning to above example, the root of the tree is the state labelled [adu] which consists of a single goal [vv6] representing the content of formula `get2`. To expand in this proof state all occurrences of the defined constants to their values, we apply the command

expand length, get, put, content;

which yields a single child state [c3b] of the following form:

Formula [get2] proof state [c3b] : scatter	
Constants (with types): anyelem, get, length, put, content, anyarray, new, any.	
d5q	$\forall a \in \text{ARR}, i \in \text{INDEX}, j \in \text{INDEX}, e \in \text{ELEM}:$ $i < a.0 \wedge j < a.0$ \Rightarrow $i = j$ \vee $\text{if } j < i < a.0 \text{ then } (a.0, a.1 \text{ with } [i]:=e) \text{ else anyarray endif } .0 \text{ then if}$ $i < a.0 \text{ then } (a.0, a.1 \text{ with } [i]:=e) \text{ else anyarray endif } .1[j] \text{ else}$ $\text{anyelem endif} = \text{if } j < a.0 \text{ then } a.1[j] \text{ else anyelem endif}$
Parent: [adu] Children: [qid]	

The state has a single goal [d5q] which is the result of the expansion of all constants in the parent's goal [vv6]. Our next task is to simplify the proof state by getting rid of the universal quantifier and of the logical connectives in the goal. The simplest way to achieve this is pressing the “Scatter State” button  which applies various proving rules in order to scatter the current state to a number of simpler ones. This yields the following dialog which indicates that the remainder of the proof has been automatically completed and that the system has returned to declaration mode:

```
Proof state [qid] is closed by decision procedure.
Formula get2 is proved. QED.
Save this proof and overwrite the previous one (y/n)? y
Proof saved (browse file get2_index.xhtml).
Quit proof of formula get2 (use 'proof get2' to see proof).
```

By selecting the command `proof get2` from the menu of formula *get2*, we see the structure of the generated proof:

```
[adu]: expand length, get, put, content
[c3b]: scatter
[qid]: proved (CVCL)
```

From state [c3b] a single child state was generated that was automatically proved by the external decision procedure CVCL. Clicking on the corresponding node in the tree displays the state as follows:

Formula [get2] proof state [qid] : proved (CVCL)	
Constants (with types): a_0 , anyelem, get, length, put, content, j_0 , new, anyarray, e_0 , any, i_0 .	
kxh	$i_0 < a_0.0$
bfe	$j_0 < a_0.0$
df4	$j_0 \neq i_0$
2ya	$\text{if } j_0 < a_0.0 \text{ then } a_0.1[j_0] \text{ else anyelem endif} = \text{if } j_0 < i_0 < a_0.0 \text{ then } (a_0.0, a_0.1$ $\text{with } [i_0]:=e_0) \text{ else anyarray endif } .0 \text{ then if } i_0 < a_0.0 \text{ then } (a_0.0, a_0.1 \text{ with } [i_0$ $]:=e_0) \text{ else anyarray endif } .1[j_0] \text{ else anyelem endif}$
Parent: [c3b]	

The state has four new constants a_0 , i_0 , j_0 , and e_0 that replace the bound variables of the universally quantified goal [d5q] in the parent state. The resulting goal without quantifier was decomposed into three atomic formulas as assumptions and a single atomic formula (the equality of two conditional expressions) as a goal. This proof state was automatically closed by the decision procedure; for a human, the corresponding reasoning steps are (although not really difficult) tedious and error-prone.

This small example already illustrates a general strategy of how to work with the system: to decompose a proof and get rid of quantifiers until sufficiently much low-level knowledge in the form of atomic predicates is available such that a decision procedure can automatically close the proof state. The task of the human (and the difficulty in real-world proofs) is to expose this low-level knowledge by guiding the overall proof construction; the task of the system is to make this process as painless as possible and to take over (via an external decision procedure) low-level reasoning on builtin datatypes.

5. A Program Verification

In this section, we are going to demonstrate the use of the RISC ProofNavigator for the verification of a Hoare triple [Hoa69]. The 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:

$$\begin{aligned} & \{ olda = a \wedge oldx = x \wedge n = |a| \wedge i = 0 \wedge r = -1 \} \\ & \mathbf{while} \ i < n \wedge r = -1 \ \mathbf{do} \\ & \quad \mathbf{if} \ a[i] = x \\ & \quad \quad \mathbf{then} \ r := i \\ & \quad \quad \mathbf{else} \ i := i + 1 \\ & \{ a = olda \wedge x = oldx \wedge \\ & \quad ((r = -1 \wedge \forall i : 0 \leq i < |a| \Rightarrow a[i] \neq x) \vee (0 \leq r < |a| \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x)) \} \end{aligned}$$

By the rules of the Hoare calculus, the verification of this triple is reduced to the proof of four verification conditions. The condition that is most difficult to prove is

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

where *Output* represents the postcondition of the above triple and *Invariant* is a suitable loop invariant:

$$\begin{aligned} Output & : \Leftrightarrow a = olda \wedge x = oldx \wedge \\ & ((r = -1 \wedge \forall i : 0 \leq i < length(a) \Rightarrow a[i] \neq x) \vee \\ & (0 \leq r < length(a) \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x)) \\ Invariant & : \Leftrightarrow olda = a \wedge oldx = x \wedge n = length(a) \wedge \\ & 0 \leq i \leq n \wedge \forall j : 0 \leq j < i \Rightarrow a[j] \neq x \wedge \\ & (r = -1 \vee (r = i \wedge i < n \wedge a[r] = x)) \end{aligned}$$

We are now going to elaborate the proof of the verification condition; this proof will ultimately have the following tree structure:

```
[dca]: expand Invariant, Output in zfg
  [tvy]: scatter
    [dca]: auto
      [t4c]: proved (CVCL)
        [ecu]: split pkg
          [kel]: proved (CVCL)
            [lel]: 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 satisfiability solver 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 $\text{Invariant}(a, x, i, n, r) \Rightarrow \text{Output} \vee i < n \wedge r = -1$

Children: [tvy]

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 [tvy] : scatter

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

aqc $\text{olda} = a \wedge \text{oldx} = x \wedge n = \text{length}(a) \wedge (\forall j \in \mathbb{N}: x = \text{get}(a, j) \Rightarrow j \geq i)$

\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

$\text{olda} = a$

\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, j₀, 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 $i \leq n$

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: [tvy] 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: [tvy] Children: [kel] [lel]

We might lack a concrete idea how to proceed any further and try our luck with `expand length`, `get`, i.e., an expansion of the definitions of the functions occurring in the state, yielding the substate [kel]:

Formula [C] proof state [kel]

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
uab	$n = a.0$
56y	$\forall j \in \mathbb{N}: x = \text{if } j < a.0 \text{ then content}(a)[j] \text{ else anyelem endif} \Rightarrow j \geq i$
2hs	$i \leq a.0$
pa2	$r = -1 \vee r = i \wedge x = \text{if } r < a.0 \text{ then content}(a)[r] \text{ else anyelem endif} \wedge i < a.0$
6tb	$r = -1 \Rightarrow a.0 \leq i$
mkm	$r = -1 \Rightarrow (\exists j \in \mathbb{N}: x = \text{if } j < a.0 \text{ then content}(a)[j] \text{ else anyelem endif} \wedge j < a.0)$
jh5	$0 \leq r$

Parent: [ecu]

However, pressing the “Auto” button  in this state also has no effect, thus we press the “Redo” button  to return to the parent state [ecu]. Since we still does not know how to proceed, we attempt the “Counterexample” button  which annotates the proof state with a “counterexample”, i.e. an interpretation of the constants which might invalidate the proof state:

Formula [C] proof state [ecu] : expand get, length

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	$0 \leq r$

This may be a counterexample:
anyarray=(1, anyarray.1) ^ length=(lambda f in [Z, array Z of ELEM]: i) ^ a=(1, a.1) ^ olda=(1, a.1)
^ oldx=x ^ i=0 ^ n=0 ^ r=-1 ^ Input ^ Output ^ i=length((1, a.1))

Parent: [tvy] Children: [kel]

Apart from some trivial equalities following from the constant definitions and proof state assumptions, the counterexample includes the formula $r = -1$ (denoting the condition “element not found” in the program being verified); this lets us suspect that its truth might be of importance. Investigating the proof state further, we realize that it contains three assumptions that start with $r = -1$. Our further reasoning depends on the fact whether this formula is true, i.e. we have to perform a corresponding case distinction. We may trigger this distinction by the command `case r=-1` or, more easily, by selecting from the menu of the formula [pkg] the command `split pkg` which generates two child states each of which receives one component of

the disjunction as an additional assumption. 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_θ , 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	$i \leq n$
gkr	$r = -1 \vee r = i \wedge x = \text{get}(a, r) \wedge i < n$
orv	$r = -1 \Rightarrow n \leq i$
k4w	$x = \text{get}(a, j_\theta)$
6ha	$j_\theta < n$
jh5	$0 \leq r$

|

Parent: [lel] Children: [lap]

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

6. Application

As mentioned in the introduction, the development of the RISC ProofNavigator has been triggered by the investigation of a number of use cases; the corresponding proofs are included in the distribution of the software. These proofs cover the following scenarios (the names set in `teletype` denote the corresponding subdirectories in the distribution).

1. Induction proofs: `sum` and `sum0`.
2. Quantifier proofs: `quant` and `quant0`.
3. Proofs based on axiomatization of arrays: `arrays1`.
4. Proofs based on constructive definition of arrays: `arrays2`.
5. Verification of linear search: `linsearch`.
6. Verification of binary search: `binarysearch`.
7. Verification of a concurrent system of one server and 2 clients: `clientServer`
8. Verification of a concurrent system of one server and N clients: `clientServerN`

Use cases 1 and 2 illustrate basic features of the system. Use case 3 gives an axiomatic definition of the datatype “array” and a proof of a fundamental array property (which becomes a bit complicated because the RISC ProofNavigator does not yet support rewriting as a proving strategy). Use cases 4–8 describe sample verifications that are suitable for classroom presentation. While use case 4 (sketched in this paper) only needs simple linear integer arithmetic, use case 5 already combines integer arithmetic with rational arithmetic and involves an axiomatization of the “floor” function mapping rationals to integers. Use cases 6 and 7 describe the verification of a mutual exclusion property of a concurrent system based on a suitable system invariant. The last two examples generate proofs that consist of some hundreds of proof states (most of which are closed automatically, the user has to apply about two dozens commands only).

With various other proving assistants, we personally got stuck somewhere between use cases 5 and 6 (which dealt for the first time with not completely trivial arithmetic properties). Only with PVS we were able to comfortably prove these use cases and also (with major effort involving several hundreds of user interactions) use case 7; the proof of use case 8 was only attempted with the RISC ProofNavigator. Compared to the corresponding PVS proofs, with the RISC ProofNavigator the proofs of use cases 7 and 8 were actually quite simple and regular; when we had devised the overall proof decomposition, the system was able to prove most of the several hundreds proof states automatically by heuristic instantiation of quantified formulas (where PVS had substantial problems).

The use cases are definitely not sufficient to qualify the RISC ProofNavigator as an industrial-strength proving assistant (as PVS certainly is). However, they demonstrate the practical usability of the software for a limited set of scenarios that are in particular typical for the use in courses on formal methods and program verification. We believe that its carefully designed interaction facilities substantially flatten the learning curve such that quicker success can be achieved within a limited time frame. Moreover, as demonstrated by some of the use cases, the software does not immediately bow to more complicated verifications and might therefore also be usable for limited non-classroom use.

As for actual classroom use, the software has been applied for the first time in the summer semester 2007 by a course on “Formal Methods in Software Development” [For07] (a regular course for students of the master studies “Software Engineering” and “Computer Mathematics” of the Johannes Kepler University Linz, Austria) replacing the previously used PVS for proving verification conditions of sequential and concurrent programs. We first introduced the software by performing sample verifications in the classroom and then gave the students similar home exercises in deriving verification conditions and proving them on their own with the help of the software.

There were virtually no complaints about the software and only few suggestions for improvement (such as the possibility to re-open already closed proof states) which we will take into account in the next version. Compared to the previous use of PVS, students seemed to get effective with the software in a shorter period; most students were quickly able to work with it and perform the verifications essentially as expected. Sometimes, however, the proofs were considerably more complicated than necessary because critical information (manual instantiations of quantified formulas or new lemmas) was not introduced early enough (close to the root of the proof tree) such that later automatic instantiations were not effective in closing proof states in the various proof branches. This is a point which certainly requires further experience and training.

Independently, the RISC ProofNavigator is at the time of writing this paper (September 2007) used in a course of the University of Waikato, New Zealand for performing proofs in a course on program reasoning [Rea07]. Furthermore, the software is going to be used from the winter semester 2007 on by another lecturer in a regular course on Formal Methods for the master program “Software Engineering” of the University of Applied Sciences in Hagenberg, Austria. Since we can thus expect the regular use of the software in the following years, we plan to investigate more systematically how students interact with it in order to further improve its usability.

7. 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. At the moment we have successfully evaluated the software by a number of use cases; soon we will also be able to report on the actual use of the software in the classroom.

Currently the major limitations of the software are the lack of rewriting as a proving strategy, i.e.

equations cannot be declared as rewriting rules which complicates proofs on e.g. axiomatically declared datatypes (rather than constructively defined ones as shown in this paper). Furthermore, the software is currently bound to a particular SMT solver (CVCL); however, a diploma thesis is under way to provide a generic interface that allows the integration of other state of the art solvers.

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

- [A⁺05] David 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>.
- [AC03] Jean-Raymond Abrial and Dominique Cansell. Click'n Prove: Interactive Proofs within Set Theory. In David A. Basin and Burkhart Wolff, editors, *TPHOLs 2003*, volume 2758 of *LNCs*, pages 1–24. Springer, 2003.
- [BB04] Clark Barrett and Sergey 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.
- [BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, Berlin, 2004.
- [BCJ⁺06] Bruno Buchberger, Adrian Craciun, Tudor Jebelean, et al. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.
- [BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.
- [Boe04] S. Buswell and others (eds). The OpenMath Standard. Version 2.0, The OpenMath Society, June 2004. <http://www.openmath.org>.
- [CMM05] Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software Refinement with Perfect Developer. In *SEFM'05: Third IEEE International Conference on Software Engineering and Formal Methods*, pages 363–373, Koblenz, Germany, September 5–9, 2005. IEEE Computer Society.
- [Fei05] Ingo Feinerer. Formal Program Verification: A Comparison of Selected Tools and Their Theoretical Foundations. Master's thesis, Theory and Logic Group, Institute of Computer Languages, Vienna University of Technology, Vienna, Austria, January 2005.
- [For07] Formal Methods for Software Development, 2007. 6 ECTS credits course, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, <http://www.risc.uni-linz.ac.at/people/schreine/courses/ss2007/formal>.
- [Gal86] Jean H. Gallier. *Logic for Computer Science — Foundations of Automatic Theorem Proving*. Harper & Row, New York, 1986.
- [Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [HR04] Michael Huth and Mark Ryan. *Logic in Computer Science — Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, UK, second edition, 2004.
- [KMM00] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
- [Koh06] Michael Kohlbase. OMDoc — An Open Markup Format for Mathematical Documents (Version 1.2). Technical Specification and Primer, Saarland University, Germany, April 2006. <http://www.mathweb.org/omdoc>.
- [NPW05] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of *Lecture Notes in Computer Science*. Springer, Berlin, 2005.
- [ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak 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.
- [Rea07] Reasoning about Programs, 2007. Course at the School of Computing and Mathematical Sciences, University of Waikato, New Zealand, <http://www.cs.waikato.ac.nz/~robi/comp340-07b/>.
- [RIS06] 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>.
- [Sch06a] Wolfgang Schreiner. Program Verification with the RISC ProofNavigator. In *Teaching Formal Methods: Practice and Experience*, Electronic Workshops in Computing (eWiC), BCS-FACS Christmas Meeting, London, UK, December 15, 2006. British Computer Society.
- [Sch06b] Wolfgang Schreiner. The RISC ProofNavigator — Tutorial and Manual. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, July 2006.
- [SMT06] SMT-LIB — The Satisfiability Modulo Theories Library, 2006. University of Iowa, Iowa City, IA, <http://combination.cs.uiowa.edu/smtlib>.
- [Wie06] Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer, Berlin, 2006.