A Comparison of PVS and Isabelle/HOL

David Griffioen[1,2]* Marieke Huisman[2]

[1] CWI, Amsterdam.
[2] Computing Science Institute, Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{marieke,davidg}@cs.kun.nl

Abstract. There is an overwhelming number of different proof tools available and it is hard to find the right one for a particular application. Manuals usually concentrate on the strong points of a proof tool, but to make a good choice, one should also know (1) which are the weak points and (2) whether the proof tool is suited for the application in hand. This paper gives an initial impetus to a consumers' report on proof tools.
The powerful higher-order logic proof tools PVS and Isabelle are compared with respect to several aspects: logic, specification language, prover, soundness, proof manager, user interface (and more). The paper concludes with a list of criteria for judging proof tools, it is applied to both PVS and Isabelle.

1994 Mathematics Subject Classification: 03B35 Mechanisation of proof and logical operations; 03B15 Higher-order logic and type-theory.
1998 Computing Reviews Classification System: F.4.3 Formal Languages; D.2.4 Software/Program Verification; F.3.1 Specifying and Verifying and Reasoning about Programs.
Keywords and Phrases: Proof Tools, Isabelle/HOL, PVS.

1 Introduction

There is an overwhelming number of different proof tools available (*e.g.* in the *Database of Existing Mechanised Reasoning Systems* one can find references to over 60 proof tools [Dat]). All have particular applications that they are especially suited for. Introductionary papers on proof tools usually emphasise their strong points by impressive examples. But, if one really wishes to start using one particular proof tool, this information is usually not enough. To make the right choice, one should also know (1) which are the weak points of the proof tool and (2) whether the proof tool is suited for the application in hand. The choice of a proof tool is very important: it can easily take half a year before one fully masters a tool and is able to work on significant applications.

It would be desirable to have some assistance in choosing the appropriate proof tool. When one wishes to buy a toaster, there is also a wide choice, but one is assisted by the reports from consumers' organisations. It is desirable to have

* Supported by the Netherlands Organisation for Scientific Research (NWO) under contract SION 612-316-125.

similar consumers' reports for proof tools. Such reports should not summarise the manuals, but they should be based on practical experience with these tools. It should discuss several important aspects from a users' perspective. These aspects should be both theoretical (*e.g.* the logic used) and practical (*e.g.* the user interface). It also should contain a list of criteria on which all proof tools are judged. This consumers' report can assist in selecting an appropriate proof tool, but it can also be interesting for people who are already using a particular proof tool (and do not have any plans to change this), because knowing about other proof tools also helps understanding the proof tool one is usually working with.

We are aware that proof tools change in time and that such a consumers' report can only have temporary validity. However, it would be nice if it could have some influence on the direction in which proof tools are developing.

This paper gives the initial impetus to such a report. It describes two proof tools, PVS [Sha96] and Isabelle [Pau94]. We have chosen PVS and Isabelle as the basis for our comparison, because both are known as powerful proof tools for higher-order logic, which have shown their capabilities in non-trivial applications. Both PVS and Isabelle are very complex tools and it is impossible to take all features into account. Therefore, our opinion on the important advantages and disadvantages of working with PVS or Isabelle, is to some extend subjective and influenced by our own histories and fields of research.

Section 1.1 briefly gives some background information on PVS and Isabelle. Next, Section 2 compares PVS and Isabelle/HOL. Section 3 discusses our experiences with PVS and Isabelle. Section 4 sketches what we think is the best of both tools. Finally, in Section 5 we apply a list of criteria to both PVS and Isabelle.

We based our experiences on PVS version 2.417 and on Isabelle versions 94-8 and 98.

Related Work We are not the first to compare different proof tools. A comparison of ACL2, a first-order logic prover based on Lisp, and PVS based on the verification of the Oral Message algorithm is described in [You97]. HOL is compared to PVS in the context of a floating-point standard [CM95]. In the first comparison, the specification language of PVS is described as too complex and sometimes confusing, while the second comparison is more enthusiastic about it. Gordon describes PVS from a HOL perspective [Gor95]. Other comparisons have been made between HOL and Isabelle/ZF (in the field of set theory) [AG95] and HOL and Coq [Zam97]. Three proof tool interfaces (including PVS) are compared from a human-computer interaction perspective in [MH96].

To the best of our knowledge, we are the first to compare PVS and Isabelle/HOL. Our comparison is not based on a particular example, but treats systematically several aspects of both tools.

1.1 Short overview of PVS and Isabelle

The **PVS** Verification System is being developed at SRI International Computer Science Laboratory. Work on PVS started in 1990 and the first version was made

available in 1993. A short overview of the history of the system can be found in [Rus]. PVS is written in Lisp and it is strongly integrated with (Gnu and X) Emacs. The source code is not freely available.

PVS has been applied to several serious problems. For example to specify and design fault-tolerant flight control systems, including a requirements specification for the Space Shuttle [CD96]. References to more applications of PVS can be found in [Rus].

Isabelle is being developed in Cambridge, UK, and in Munich. The first version of the system was made available in 1986. Isabelle uses several ideas of the LCF prover [GMW79]: formulae are ML values, theorems are part of an abstract data type and backward proving is supported by tactics and tacticals. The aim of the designers of Isabelle was to develop a generic proof checker, supporting a variety of logics, with a high level of automation. Isabelle has been called the *next 700 provers* [Pau90]. Isabelle is written in ML, and the source code is freely available.

Isabelle is used in a broad range of applications: formalising mathematics (including semantics), logical investigations, program development, specification languages, and verification of programs or systems. References to applications of Isabelle can be found in [Pfe].

2 A comparison of PVS and Isabelle/HOL

This section first describes several important aspects of a proof tool in general. The comparison of PVS and Isabelle will then be structured along these lines. The division is somewhat artificial, because strong dependencies exist between the various parts, but is helpful in the comparison. The emphasis will be on aspects that are important from a users' perspective

The first aspect that we distinguish is the **logic** that is used by the tool. In this paper we will restrict ourselves to (extensions of) typed higher-order logic.

Strongly related with the logic is the **specification language**. It is very important to have a good specification language, because a significant part of a verification effort comes down to specifying what one actually wishes to verify. It is not very useful to have a fully verified statement, if it is not clear what the statement means.

The next aspect that we distinguish is the **prover**. An important issue for the prover is which proof commands (tactics) are available (*i.e.* which steps can be taken in a proof). Strongly related with this is the choice of a **tactical language**. Tacticals or proof strategies are functions which build new proof commands, using more basic ones. A sophisticated tactical language significantly improves the power of a prover. Another important aspect is whether **decision procedures** (such as for linear arithmetic and for abstract data types) are available.

A next aspect is the structure of the tool, *i.e.* whether there is a small kernel which does all logical inferences. When the code of the kernel is available (and small) it is possible to convince oneself of the **soundness** of the tool.

Another component is the **proof manager**, which determines *e.g.* how the current subgoals are displayed, whether the proof trace is recorded and how proof commands can be undone.

Theoretically non-existent, but very important for the actual use of a tool, is the **user interface**. Of course this does not influence the "computing power" of the tool, but a good user interface can significantly increase the effectiveness and usability of a proof tool.

2.1 The logic

PVS PVS implements classical typed higher-order logic, extended with predicate subtypes and dependent types. PVS has many built-in types, such as booleans, lists, reals and integers; standard operations on these types are also hard-coded in the tool. Type constructors are available to build complex types *e.g.* function types, product types, records (labelled products) and recursively-defined abstract data types. The use of predicate subtypes and dependent types will be explained in more detail below.

Isabelle Isabelle has a meta-logic, which is a fragment of higher-order logic. Formulae in the meta-logic are build using implication $\Rightarrow$, universal quantification $\bigwedge$ and equality $\equiv$. All other logics (the object logics) are represented in this meta-logic. Examples of object logics are first-order logic, the Barendregt cube, Zermelo-Fraenkel set theory and (typed) higher-order logic.

In this paper we will restrict attention to typed higher-order logic (HOL) as object logic. The formalisation of HOL in Isabelle relies heavily on the meta-logic. HOL uses the polymorphic type system of the meta-logic. In its turn, the type system of the meta-logic is similar to the type system of ML, the implementation language. Implication, quantification and equality are immediately defined in terms of the meta-logic. Together with some appropriate axioms, these form the basis for the higher-order logic theory. All other definitions, theorems and axioms are formulated in terms of these basic constructs.

Predicate subtypes and dependent types Predicate subtypes and dependent types as in PVS are not common in mechanical proof checkers, but they can be very useful in writing down a succinct and correct specification.

A predicate subtype is a new type constructed from an existing type, by collecting all the elements in the existing type that satisfy the predicate. Perhaps, the most famous basic example of a predicate subtype is the type of non-zero-numbers. This type is used in the declaration of the division operator in PVS. The code below[1] is a fragment of the PVS prelude (which contains the theories that are built-in to the PVS system).

```
nonzero_real: NONEMPTY_TYPE = {r: real | r /= 0}    % /= is inequality

+, -, * : [real, real -> real]
/ : [real, nonzero_real -> real]
```

[1] All examples in this paper are available at
`http://www.cs.kun.nl/ marieke/Comparison.html`.

```
Ex_Array[T:TYPE]: THEORY
BEGIN
  Ex_Array: TYPE = [# length : nat,
                      val : [below(length) -> T ]
                   #]
END Ex_Array
```

Fig. 1. Dependent typing in PVS

When the division operator is used in a specification, type checking will require that the denominator is nonzero. As this is not decidable in general, a so-called Type Correctness Condition (TCC) is generated, which forces the user to prove that the denominator is indeed nonzero. A theory is not completely verified unless all of its type correctness conditions have been proven. In practice, most of the TCCs can be proven automatically by the tool. The use of predicate subtypes improves the readability of a specification and helps in detecting many semantical errors, as the user can state explicitly all the type constraints. Carreño and Miner come to the same conclusion in [CM95].
As mentioned, PVS offers another typing facility namely dependent typing. In Figure 2.1 a theory of arrays is depicted. The type `Ex_Array` is a record with two fields: `length` a natural number denoting the length of the array, and `val` a function denoting the values at each position in the array. The domain of `val` is the predicate subtype `below(length)` of the natural numbers less than `length`. The type of `val` thus depends on the actual length of the array[2].

2.2 The specification language

PVS The specification language of PVS is rich, containing many different type constructors, predicate subtypes and dependent types. As an example, a specification of the quicksort algorithm can be found in Figure 2. We discuss some specific points.

- PVS has a **parametrised module** system. A specification is usually divided in several theories and each theory can be parametrised with both types and values. Theories can import (multiple) other theories from every point in the theory, so that a value or type that has just been declared or defined can immediately be used as an actual parameter.
  Polymorphism is not available in PVS, but it is approximated by theories with type parameters. To define a polymorphic function, one can put it in a theory which is parametrised with the type variables of the function. However, this approach is not always convenient, because when a theory is imported *all* parameters should have a value, thus when a function does not use all type parameters of a theory, the unused types should still get some instantiation.

[2] Dependent typing and predicate subtyping in general are separate matters, but in PVS dependent types can only be constructed using predicate subtypes.

```
sort[T:TYPE,<=:[T,T->bool]]: THEORY % parametrised theory
BEGIN

 ASSUMING                              % assuming clause
   total: ASSUMPTION total_order?(<=)    % infix operator
 ENDASSUMING

 l : VAR list[T]
 e : VAR T

 sorted(l): RECURSIVE bool =          % recursive definitions
    IF null?(l) OR null?(cdr(l))      % with measure
    THEN true
    ELSE car(l) <= car(cdr(l)) AND sorted(cdr(l))
    ENDIF
 MEASURE length(l)

 qsort(l): RECURSIVE list[T] =
    IF null?(l) THEN null
    ELSE LET piv = car(l)
         IN append(qsort(filter(cdr(l),(LAMBDA e: e <= piv))),
                   cons(piv,
                   qsort(filter(cdr(l),(LAMBDA e: NOT e <= piv)))))
    ENDIF
  MEASURE length(l)

 qsort_sorted: LEMMA sorted(qsort(l))

END sort
```

Fig. 2. A specification of the quicksort algorithm in PVS

- PVS has a rich **overloading** structure. Different functions can have the same name as long as they have different input types. Different functions in different theories can have the same name, even when they have the same (input) type. The theory name can be used as a prefix to distinguish between them. Names for theorems and axioms can be reused as well, as long as they are in different theories. Again, the theory name can be used to disambiguate this.
- A theory can start with a so-called **assuming clause**, where one states assumptions, usually about the parameters of the theory. These assumptions are used as a fact in the rest of the theory. When the theory is imported, TCCs are generated, which force the user to prove that the assumptions hold for the actual parameters.
- **Recursive data types and functions** can be defined in PVS. An induction principle and several standard functions, such as map and reduce, are automatically generated from an abstract data type definition. PVS allows general recursive function definitions. All functions in PVS have to be total,

```
QSort = HOL + List + WF_Rel +      (* theory importings *)

consts                             (* infix operators *)
  "<<=" :: "['a, 'a] => bool"          (infixl 65)

axclass                            (* axiomatic type class *)
  ordclass < term
  total_ord "total (op <<=)"

consts                             (* primitive recursion *)
  sorted:: "[('a :: ordclass) list] => bool"
primrec sorted list
  sorted_nil  "sorted [] = True"
  sorted_cons "sorted (x#xs) = ((case xs of [] => True | y#ys => x <<= y) &
                                sorted xs)"

consts                              (* well-founded recursion *)
  qsort :: "[('a :: ordclass) list] => ('a :: ordclass) list"
recdef
  qsort "measure size"
    "qsort [] = []"
    "qsort (x # xs) = qsort [y : xs. y <<= x] @
                            (x # qsort [y : xs. ~ y <<= x])"

end
```

Fig. 3. A specification of the quicksort algorithm in Isabelle

therefore termination of the recursive function has to be shown, by giving a measure function which maps the arguments of the function to a type with a well-founded ordering. The tool generates TCCs that force the user to prove that this measure decreases with every recursive call.

- PVS has much fixed **syntax**. Many language constructs, such as `IF...` and `CASES...` are built-in to the language and the prover. There is a fixed list of symbols which can be used as infix operators; most common infix operators, such as `+` and `<=` are included in this list. Sometimes PVS uses syntax which is not the most common, *e.g.* `[A,B]` for a Cartesian product of types `A` and `B` and `(:x,y,z:)` for a list of values `x`,`y`,`z`.

Isabelle The specification language of Isabelle is inspired by functional programming languages (especially ML). In Figure 3 the quicksort example is shown in Isabelle syntax. We discuss some specific aspects.

- The **module system** allows importing multiple other theories, but it does not permit parametrisation. The type parameters of PVS are not necessary in Isabelle, because functions can be declared polymorphically. The value parameters of PVS can be thought of as an implicit argument for all functions in the theory. Making this argument explicit could be the way to 'mimic' the value parameters in Isabelle.

- **Axiomatic type classes** [Wen95,Wen97] are comparable to the assuming clause in PVS, and type classes in functional programming [WB89]. In a type class polymorphic declarations for functions are given. Additionally, in *axiomatic* type classes required properties about these functions can also be stated. These properties can be used as axioms in the rest of the theory. The user can make different instantiations of these axiomatic type classes, by giving appropriate bodies for the functions and proving that the properties hold. Notice that a limited form of overloading can be realised using Isabelle's axiomatic type classes, only for functions with a single polymorphic type.
- Isabelle automatically generates induction principles for each **recursive data type**. The user can give **inductive** and **coinductive** function definitions. There is a special construct to define primitive recursive functions. Well-founded recursive functions can be defined as well, together with a measure function to show their termination.
- Isabelle **syntax** can easily be extended. In particular, Isabelle allows the user to define arbitrary infix and mixfix operators. There is a powerful facility to give priorities and to describe a preferred syntax. This allows the user to define that lists should be represented for input and output as *e.g.* [1,2,3] while internally this is represented as `(cons 1 (cons 2 (cons 3 nil)))`. Language constructs like `if...then...else` are defined explicitly in terms of the basic operators.

2.3 The prover

PVS PVS represents theorems using the sequent calculus. Every subgoal consists of a list of assumptions $A_1, \ldots A_n$ and a list of conclusions $B_1, \ldots, B_m$. One should read this as: the conjunction of the assumptions implies the disjunction of the conclusions *i.e.* $A_1 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee \ldots \vee B_m$.

The proof commands of PVS can be divided into three different categories[3].

- **Creative proof commands.** These are the proof steps one also writes down explicitly when writing a proof by hand. Examples of such commands are `induct` (start to prove by induction), `inst` (instantiate a universally quantified assumption, or existentially quantified conclusion), `lemma` (use a theorem, axiom or definition) and `case` (make a case distinction). For most commands, there are variants which increase the degree of automation, *e.g.* the command `inst?` tries to find an appropriate instantiation itself.
- **Bureaucratic proof commands.** When writing a proof by hand, these steps usually are done implicitly. Examples are `flatten` (disjunctive simplification) `expand` (expanding a definition), `replace` (replace a term by an equivalent term) and `hide` (hide assumptions or conclusions which have become irrelevant).
- **Powerful proof commands.** These are the commands that are intended to handle all "trivial" goals. The basic commands in this category are `simplify`

[3] This division is made by the authors, not by the developers of PVS. Nevertheless it resembles the division made in [COR+95].

and `prop` (simplification and propositional reasoning). A more powerful example is `assert`. This uses the simplification command and the built-in decision procedures and does automatic (conditional) rewriting. PVS has some powerful decision procedures, dealing, among other things, with linear arithmetic. The most powerful command is `grind`, which unfolds definitions, skolemizes quantifications, lifts if-then-elses and tries to instantiate and simplify the goal.

Isabelle The basic proof method of Isabelle is resolution. The operation `RS` is the standard resolution operation. It unifies the conclusion of its first argument with the first assumption of the second argument. As an example, when doing resolution with $([\![?P]\!] \Rightarrow ?P \vee ?Q)$ and $([\![?R; ?S]\!] \Rightarrow ?R \wedge ?S)$, this results in the theorem $[\![?P; ?S]\!] \Rightarrow (?P \vee ?Q) \wedge ?S$.

Isabelle supports both forward and backward proving, although its emphasise lies on backward proving by supplying many useful tactics for it. A tactic transforms the proof goal into several subgoals and gives a justification for this transformation.

In Isabelle, every goal consists of a list of assumptions and one conclusion. The goal $[\![A_1; A_2; \ldots; A_n]\!] \Rightarrow B$ should be read as $A_1 \Rightarrow (A_2 \Rightarrow \ldots (A_n \Rightarrow B))$. Notice that $\Rightarrow$ is the implication of the meta-logic.

Isabelle tactics usually do not return a single next state, but a lazy list with possible next states. Many tactics try to find a useful instantiation themselves and return a lazy list containing (almost) all possible instantiations (in a suitable order). When the first instantiation is not satisfactory the next instantiation can be tried with `back`. This possibility is mainly used by powerful tactics.

The proof commands of Isabelle can be divided in several categories as well, although these are different from the categories used earlier for PVS.

- **Resolution** is the basis for many tactics. The standard one is `resolve_tac`. It tries to unify the conclusion of a theorem with the conclusion of a subgoal. If this succeeds, it creates new subgoals to prove the assumptions of the theorem (after substitution).
- Another basic tactic is `assume_tac`, which tries to unify the conclusion with one of the assumptions.
- **Induction** is done by `induct_tac`, which does resolution with an appropriate induction rule.
- **Use an axiom or theorem** by adding it to the assumption list. There are several variants: with and without instantiation, in combination with resolution etc.
- **Simplification** tactics for (conditional) rewriting. For every logic a so-called simplification set can be build. This set contains theorems, axioms and definition, that can be used to rewrite a goal. It is possible to extend the simplification set (temporarily or permanent).
  Isabelle's simplifier uses a special strategy to handle permutative rewrite rules, *i.e* rules where the left and right hand side are the same, up to renaming of variables. A standard lexical order on terms is defined and a

permutative rewrite rule only is applied if this decreases the term, according to this order. The most common example of a permutative rewrite rule is commutativity ($x \oplus y = y \oplus x$). With normal rewriting (as is done by PVS) this rule will loop, but ordered rewriting avoids this.

- **Classical reasoning** is another powerful proof facility of Isabelle. There are various tactics for classical reasoning. One of them, `blast_tac`, uses a tableau prover, coded directly in ML. The proof that is generated is then reconstructed in Isabelle.
- **Bureaucratic** tactics are also available, such as `rotate_tac`, which changes the order of the assumptions. This can be necessary for rewriting with the assumptions, because this is done from top to bottom.

A theorem can contain so-called meta-variables, which can be bound while proving it. As an example, consider the specification of quicksort (Figure 3). Suppose that we instantiated the axiomatic type class with the natural numbers (defining `<<=` as $\leq$) and that the definition of quicksort is automatically rewritten. Now we can state for example the following goal

```
goal QSort.thy "qsort[4, 2, 3] = ?x";
```

where `?x` is a meta-variable. When simplifying this goal, the meta-variable is bound to `[2,3,4]` (and the theorem is proven). The theorem is stored as `qsort[4, 2, 3] = [2, 3, 4]` This feature makes Isabelle well-suited for transformational programming [AB96] and writing a Prolog interpreter [Pau94].

Tactical language A tactical (or proof strategy) is a function to build complex tactics (or proof commands) using more basic ones. A well-known example is the tactical `then`. This tactical gets two tactics as arguments and applies them sequentially to the goal.

PVS has a very limited proof strategy language; roughly it is only possible to concatenate and repeat proof commands in several ways. When one wishes to go beyond this, for example to inspect the goal, this should be done in Lisp. The Lisp data structure that contains the proof goal is not officially documented; some accessor functions are known to work but the developers explicitly allow themselves to change PVS at this level of implementation. Probably it is possible to change the goal in Lisp without a logical justification.

In Isabelle the tactical language is ML, so a complete functional language is available. All logical inferences on terms of type `thm` (the theorems) are performed by a limited set of functions. In ML a type can be 'closed', which means that a programmer can express that no other functions than a number of 'trusted' functions are allowed to manipulate values of this type (in this case: theorems). In this way the full power of ML can be used to program proof strategies, and soundness is guaranteed via the interface.

Proving with powerful proof commands Both PVS and Isabelle can do simple calculations quite fast. For instance the theorem below is proven in (almost) zero time in PVS by `(ASSERT)`, using the built-in integer arithmetic.

```
calc: LEMMA 700 * 400 * 11 = 2 * 7 * 22 * 10000
```

In Isabelle/HOL we have a similar result. After loading the theories defining the integers we can prove the following goal in (almost) zero time using simplification. Note that integers have a sharp-sign # as prefix. Operations on integers are defined using their binary representation, so in contrast to PVS, arithmetic is not part of the kernel, but defined in the logic.

```
goal Bin.thy "#700 * #400 * #11 = #2 * #7 * #22 * #10000";
```

Linear (and some non-linear) arithmetic has standard support in PVS and the next theorem is also proven with a single command.

```
arith: LEMMA 7 + x < 8 + x  AND 2 * x * x <= 3 * x * x
```

In Isabelle a package to cancel out common summands (and factors) is available. It is loaded standardly for the naturals, but not for the integers. The following goal is proven in one step, using simplification.

```
goal Arith.thy "1 + x < 2 + x";
```

A well-known [COR+95] example of the simplification procedures of PVS is the proof of the characterisation of the summation function. The theorem below is proven by a single command `(induct-and-simplify "k")`

```
sum(k:nat): RECURSIVE nat =
  IF k = 0 THEN 0 ELSE k + sum(k-1) ENDIF
MEASURE k

sum_char: LEMMA sum(k) = k*(k+1)/2
```

An impressive example of the classical reasoner of Isabelle is the following theorem, problem 41 of Pelletier. PVS can not prove this in one command, while Isabelle can, using the classical reasoner (`Blast_tac`).

```
(ALL z. EX y. ALL x.  J x y = (J x z & (~ J x x))) --> ~(EX z. ALL x. J x z)
```

2.4 System organisation and soundness

PVS The developers of PVS designed their prover to be useful for real world problems. Therefore the specification language should be rich and the prover fast with a high degree of automation. To achieve this, powerful decision procedures were added to PVS. However, these decision procedures sometimes cause soundness problems, thus the procedures are part of the kernel, which makes the kernel large and complex. Further, PVS once was considered to be a prototype for a new SRI prover. Perhaps for these reasons PVS still seems to contain a lot of bugs and frequently new bugs shows up. An overview of the known bugs at the moment can be seen on `http://www.csl.sri.com/htbin/pvs/pvs-bug-list`. It would be desirable that the bugs in PVS would only influence completeness and not soundness. Unfortunately, this is not the case, as some recent proofs of

true=false have shown [Owr]. Most bugs do not influence soundness, but they can be very annoying.

Because of the soundness bugs in the past, it is reasonable to assume that PVS will continue to contain soundness bugs. The obvious question thus arises, why use a proof tool that probably contains soundness bugs? Our answer is threefold:

PVS is still a very critical reader of proofs. PVS lets fewer mistakes slip through than many of our human colleagues (and PVS is much more patient), thus in comparing PVS to an average logician/mathematician PVS is much more precise and sceptic.

Furthermore, history tells us that the fixed soundness bugs are hardly ever unintentionally explored, we know of only a single case.

Thirdly, most mistakes in a system that is to be verified are detected in the process of making a formal specification. Thus economically spoken, the specification is very important, and PVS has a expressive and human friendly specification language. Therefore when we specify a system in the language of PVS this gives extra confidence that the specification expresses what is 'meant'.

A lot of effort has been put into the development of PVS. For this reason SRI does not make the code of PVS freely available. As a consequence, to most users the structure of the tool is unknown and making extensions or bug fixes is impossible, although sometimes users go to SRI to implement a feature.

Isabelle Isabelle was developed from quite a different perspective. The main objective was to develop a flexible and sound prover, and next to develop powerful tactics, so that large proof steps could be taken at once. Isabelle seems to be much more stable than PVS. It does not show unpredictable behaviour. Recently a new Isabelle version was released[4]. To our surprise some tactics (especially `Auto_tac`) were changed, so that our old proofs really had to be adapted, and not all of these changes were clearly documented.

2.5 The proof manager

PVS All proofs in PVS are done in a special proof mode. The tool manages which subgoals still have to be proven and which steps are taken to construct a proof, so it is not the users responsibility to maintain the proof trace. Proofs are represented as trees. There is an Tcl/Tk interface which gives a picture of the proof tree (see Figure 4). It helps the user to see which branches of the proof are not proven yet. One can click on a turnstile to see a particular subgoal, also the proof commands can be displayed in full detail.

When using a proof tool most of the time the theorems and specification are under construction, as the processes of specifying and proving are usually intermingled. The notion of "unproved theorem" allows to concentrate on the crucial theorems first and prove the auxiliary theorems later. PVS keeps track of the status of proofs, e.g. whether it uses unproved theorems.

[4] Isabelle98

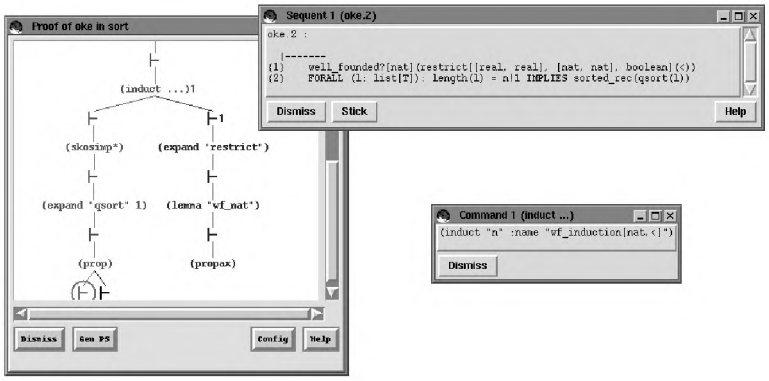

Fig. 4. Example of a Tcl/Tk proof tree

Line numbers can be used in PVS to specifiy that a command should work only on some of the assumptions/conclusions, *e.g.* `(expand "f" 2)` expands `f` in the second conclusion. When a specification or theorem is slightly changed (*e.g.* an conjuct is added), the line numbers in the goal often change. It would be more robust, if one could use commands expressing things like: expand all `f`s with zero as first argument, and only expand `f` in the assumptions where function g occurs. This has an additional advantage, namely that intention of the proof step becomes clearer. The authors have made their own Lisp functions to calculate a list of line numbers that satisfy a simple regular expression. This is already helpful (espcially in strategies), but many extensions are possible. For example, in the presence of overloading it would be useful to expand `f`s of a specific type.

Isabelle Isabelle does not give elaborate proof support. The user has to keep track of everything him/herself (including the undos). The proofs are structured linearly, there is just a list of all subgoals. This stimulates the use of tacticals such as `ALLGOALS`, but it is not so easy to see how "deep" or in which branch one is in a proof. On the other hand, in Isabelle it is possible to undo an undo (or actually: a choplev, which steps back an arbitrary number of levels, or to a particular level). And even more, it is also possible to look at the subgoals at an earlier level, without undoing the proof.

2.6 User interface

PVS's standard user interface is better developed than Isabelle's. It is is strongly integrated with Emacs. Recently, a batch mode was added to PVS. The *de facto* interface for Isabelle is Isamode (also based on Emacs). There are some more advanced user interfaces based on Tcl/Tk, but they only work for particular versions of Isabelle.

2.7 Manuals and support

PVS has a number of different manuals, but none of these is completely up-to-date. There is an introductionary manual with a fully elaborated (non-trivial) example to get started. On the mailing list one can ask starters questions.

Isabelle also comes equipped with several manuals. These are more up-to-date and concise, but often they explain things very briefly (and sometimes cryptic). The introductionary manual does not really give an interesting example, and it is hard to start using Isabelle, only on the basis of the manuals. The best way to start is to take the (annual) Isabelle course. There is good (personal) support from the developers. They usually reply very quickly (same day) on emails with questions and problems. We found that this was really helpful.

2.8 Runtime speed

We did not compare the speed of the tools because we think the game is not to "run" a proof, but to construct it. This construction consists of building a specification of a problem and proving appropriate theorems. This is hard and depends heavily on the user, his/her experience with the proof tool etc. We do mention though that the "experienced speed" of the two tools is comparable. By this we mean the time it takes to type check a specification or to execute a smart tactic.

3 Our experiences

In this section we wish to discuss in some detail our own, more personal, experiences. After using PVS for several years we became increasingly unhappy with it, because so many bugs appeared. Sometimes it felt that we would spend more time on working around small bugs, than on proving serious properties. In this period the first author visited Munich and became enthusiastic about Isabelle. However, reading the Isabelle manuals did not provide enough background to get really started with it. Therefore, in September 1997 the second author visited the Isabelle course in Cambridge. After this course, it seemed relatively easy to start working seriously with Isabelle.

To start with a well-understood, but non-trivial example, the *Tree Identification Phase (TIP)*[DGRV97] of the 1394 protocol was selected, as the first author had already worked extensively on it using PVS. The first challenge was to transform the PVS specification into Isabelle, because Isabelle's specification language lacks *e.g.* records and function updates.

The next step was to start proving. We are used to PVS's proof manager, which records all the steps we take in a proof. Isabelle only provides a so-called *listener*, which records everything the user types in (including the typos and steps that were undone later), so the proof has to be filtered out. We experienced that it works faster to copy the steps immediately than to use the listener.

When we then really started proving, we noticed a big difference in the handling of conditional expressions (*i.e.* `if...then...else`). In PVS, conditionals

are built-in and the prover knows how to deal with them. In Isabelle conditional expressions are explicitly defined and the prover does not have special facilities for them. We discussed this with Larry Paulson and Tobias Nipkow, which resulted in a solution for Isabelle94-8. In Isabelle98 more tactics to deal with conditional expressions are standard available.

After proving some invariants over the *TIP* protocol, we also studied whether a translation of object-oriented specifications into higher-order logic (part of a different project [HHJT98]) could be adapted to Isabelle. In the translation to PVS we made extensive use of overloading and this caused serious difficulties. In discussions with the Isabelle developers we tried several solutions, but none of these were satisfactory. Isabelle98 has the possibility to define different name spaces and this might help. Due to time constraints and lack of documentation we did not investigate this option.

4 The best of both worlds

When comparing PVS and Isabelle we realised that both tools had their advantages and disadvantages. Our ideal proof tool would combine the best of both worlds.

The logic Predicate subtyping and dependent typing give so much extra expressiveness and protection against semantical errors, that this should be supported. The loss of decidability of type checking is easily (and elegantly) overcome by the generation of TCCs and the availability of a proof checker.

The meta-logic of Isabelle gives the flexibility to use different logics, even in a single proof. However, in our applications, we did not feel the need to use a logic other than HOL and the interference with the meta-logic sometimes complicated matters.

The specification language The specification language should be readable, expressive and easily extendible. For function application, we have a slight preference for the bracketless syntax of Isabelle.

It should be possible to parametrise theories with values. We have a preference for type parametrised theories, because polymorphism is hard to combine with overloading. A disadvantage of type inference, in combination with implicitly (universally) quantified variables, is that typos introduce new variables, and do not produce an error. As an example, suppose that one has declared a function `myFunction :: nat => nat`, but that by accident the following goal is typed in: `"myFunction x < myFuntion (x+1)"`. This is internally equivalent to: `"ALL myFuntion. myFunction x < myFuntion (x+1)"`. This error can only be detected by asking explicitly for the list of variables (and their types) in the goal.

The prover The ideal prover has powerful proof commands for classical reasoning and rewriting, including ordered rewriting. A tactic should return a lazy list of possible next states, as this is useful to try (almost) all possible instantiations.

Also, decision procedures (for example for linear arithmetic) should be available. Preferably, these decision procedures are not built-in to the kernel, but written in the tactical language, so that they can not cause soundness problems. The style of the interactive proof commands of PVS is preferred over that of Isabelle, because this is more intuitive. It is important to have a structured tactical language, which allows the user to access the goal. For this purpose, the structure of the goal should be well-documented.

System organisation To ensure soundness of the proof tool, the system should have a small kernel. The code of the tool should be freely available, so that users can easily extend it for their own purposes and (if necessary) implement bug fixes.

The proof manager and user interface The tool should keep track of the proof trace. Proofs are best represented as trees, because this is more natural, compared to a linear structure. The tree representation also allows easy navigation through the proof, supported by a visual representation of the tree. When replaying the proof, after changing the specification, the tool can detect for which branches the proof fails, thanks to the tree representation.

5 Conclusions and future work

We tried to describe some important aspects of PVS and Isabelle which are not in the 'advertising of the tool', but are important in making a decision on which tool to use. To conclude, Figure 5 gives a list of criteria for judging a proof tool, filled in for PVS and Isabelle. This list is not complete and based on the available features of PVS and Isabelle and our work done with these proof tools. We hope that in the future users of other proof tools will produce a similar consumers' test on "their" proof tool too, so that a broad overview of users' experiences with different proof tools will be available.

Maybe such comparisons will lead to a proof tool which combines the best of all available proof tools. Looking only at PVS and Isabelle, it would be desirable to have a proof tool with the specification language, proof manager and user interface of PVS, but the soundness, flexibility and well-structuredness of Isabelle.

Acknowledgements

We thank Bart Jacobs and Frits Vaandrager for their comments on earlier drafts of this paper.

References

[AB96] Abdelwaheb Ayari and David A. Basin. Generic system support for deductive program development. In T. Margaria and B. Steffen, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and*

	PVS 2.417	Isabelle98/HOL
logic	typed HOL	typed HOL
dependent types	++	not available
predicate subtypes	++	not available
standard syntax	++/+	+
flexible syntax	-	++
module system	++/+	+
polymorphism	-	++
overloading	++	-
abstract data types	++/+	++/+
recursive functions	++/+	++/+
proof command language	+	+/-
tactical language	-	++
automation	+	+
arithmetic decision procedures	++	+/-
libraries	+	++/+
proof manager	++	+/-
interface	++	+
soundness	-	++
upwards compatible	+/-	+/-
easy to start using	+	-
manuals	+/-	+/-
support	+	++
time it takes to fix a bug	-	?
ease of installation	++	++

Fig. 5. A consumer report of PVS and Isabelle

Analysis of Systems, Passau, Germany, volume 1055 of *LNCS*. Springer-Verlag, April 1996.

[AG95] Sten Agerholm and Mike Gordon. Experiments with ZF set theory in HOL and Isabelle. In E. Thomas Schubert, Philip J. Windley, and James Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications,* Aspen Grove, UT, USA, volume 971 of *LNCS*. Springer-Verlag, September 1995.

[CD96] Judith Crow and Ben L. Di Vito. Formalizing Space Shuttle software requirements. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 40–48, San Diego, CA, January 1996. Association for Computing Machinery.

[CM95] Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *HOL95: Eighth International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, September 1995. Category B proceedings, available at `http://lal.cs.byu.edu/lal/hol95/Bprocs/indexB.html`.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[Dat] Database of existing mechanized reasoning systems. `http://www-formal.stanford.edu/clt/ARS/systems.html`.

[DGRV97] Marco Devillers, David Griffioen, Judi Romijn, and Frits Vaandrager. Verification of a leader election protocol formal methods applied to IEEE 1394. Technical Report CSI-R9728, Computing Science Institute, Catholic University of Nijmegen, 1997.

[GF97] Elsa L. Gunter and Amy Felty, editors. *Proceedings of the 10th International Workshop on Theorem Proving in Higher Order Logics,* Murray Hill, NJ, USA, volume 1275 of *LNCS*. Springer-Verlag, August 1997.

[GMW79] Michael J.C. Gordon, Robin Milner, and Cristopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[Gor95] Mike Gordon. Notes on PVS from a HOL perspective. Available at `http://www.cl.cam.ac.uk/users/mjcg/PVS.html`, August 1995.

[HHJT98] Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *Proceedings of ESOP at ETAPS '98*, LNCS. Springer-Verlag, 1998. To appear.

[MH96] Nicholas A. Merriam and Michael D. Harrison. Evaluating the interfaces of three theorem proving assistants. In F. Bodart and J. Vanderdonckt, editors, *Proceedings of the 3rd International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems*, Eurographics Series, Namur, Belgium, June 1996. Springer-Verlag.

[Owr] Sam Owre. `http://www.csl.sri.com/htbin/pvs/pvs-bug-list`
Bug numbers: 71, 82, 113 and 160.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Pfe] Frank Pfenning. Isabelle bibliography. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/biblio.html`.

[Rus] John Rushby. PVS bibliography. `http://www.csl.sri.com/pvs-bib.html`.

[Sha96] N. Shankar. PVS: Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*, pages 257–264, Palo Alto, CA, November 1996. Springer-Verlag.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[Wen95] Markus Wenzel. Using axiomatic type classes in Isabelle, a tutorial, 1995. `http://www4.informatik.tu-muenchen.de/~wenzelm/papers.html`.

[Wen97] Markus Wenzel. Type classes and overloading in higher-order logic. In Gunter and Felty [GF97].

[You97] William D. Young. Comparing verification systems: Interactive Consistency in ACL2. *IEEE Transactions on Software Engineering*, 23(4):214–223, April 1997.

[Zam97] Vincent Zammit. A comparative study of Coq and HOL. In Gunter and Felty [GF97].