Towards Tool Support for Program Verification *and* Construction

Richard Verhoeven* and Roland Backhouse

Department of Mathematics and Computing Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands
{river,rolandb}@win.tue.nl

Abstract. Math/pad is a document preparation system designed and developed by the authors and oriented towards the calculational construction of programs . PVS (Prototype Verification System) is a theorem checker developed at SRI that has been extensively used for verifying software, in particular in safety-critical applications. This paper describes how these two systems have been combined into one. We discuss the potential benefits of the combination seen from the viewpoint of someone wanting to use formal methods for the construction of computer programs, and we discuss the architecture of the combined system for the benefit of anyone wanting to investigate combining the Math/pad system with other programming tools.

Contact Author: Richard Verhoeven
Phone: +31-402475145
Fax: +31-402463992
URL: http://www.win.tue.nl/cs/wp/
E-mail: river@win.tue.nl
Keywords: tools, theorem proving, proof checkers, notations

* Research supported by the Dutch Organisation for Scientific Research (NWO) under contract SION 612-14-001

1 Introduction

Math∫pad [5] is a document preparation system designed and implemented by the first author under the direction of the second author, initially with the help of Olaf Weber. The almost-WYSIWYG nature and flexibility of Math∫pad means that it can be used for on-screen mathematical calculation (in any formal system) and, in particular, for the calculational construction and documentation of programs, this being indeed the purpose for which the system was originally designed. The system has now been stable for several years and has been used to write a number of Ph.D. and M.Sc. theses and articles in the area of the mathematics of program construction [3] and program specification using Z [8], as well as the on-line documentation of the system itself [6].

PVS (Prototype Verification System) is a theorem checker developed at SRI that has been extensively used for verifying software, in particular in safety-critical applications. A description of PVS is given on the "What is PVS?" page at SRI [17]:

> PVS is a verification system: that is, a specification language integrated with support tools and a theorem prover. It is intended to capture the state-of-the-art in mechanized formal methods and to be sufficiently rugged that it can be used for significant applications. PVS is a research prototype: it evolves and improves as we develop or apply new capabilities, and as the stress of real use exposes new requirements.
>
> PVS is a large and complex system and it takes a long while to learn to use it effectively. You should be prepared to invest six months to become a moderately skilled user (less if you already know other verification systems, more if you need to learn logic or unlearn Z)

Math∫pad and PVS have completely different design goals, stemming from the fact that Math∫pad is intended to support the (formal) *construction* of computer programs, whereas PVS is designed to support the *verification* of existing programs. Thus Math∫pad supports the language of mathematics, in its full generality, whereas PVS constrains its user to its own ASCII-based teletype language. But Math∫pad does not purport to validate or verify the user's calculations in any way, that being the responsibility of the user, whereas PVS does.

The design of Math∫pad reflects what we believe to be the highest priorities in developing tools to support the use of formal methods for software design. Above all, we concur wholeheartedly with Knuth's view [12] that programming is best viewed as a document preparation activity, the documentation serving to integrate the many different aspects (requirements, specification, implementation, testing etc.) of a highly complex process. Furthermore the language of programming specification is the language of mathematics — in other words, precise and concise, but unconstrained and subject to continual evolution and adaptation. Finally, the goal of formal methods is to ensure that programs are *correct by construction*, i.e. that the discipline of programming guarantees (when applied conscientiously and correctly) that the constructed program satisfies its specification.

This is not to say that program verification is not important. Independent checks on the validity of computer programs are vital to reliability guarantees and quality control. Formal verification, model checking, extensive (manual) testing and (independent) code walk-throughs all contribute in their own way, and none should be neglected in the real world of software design, particularly where safety is significant. But program verification can only be truly helpful if it doesn't require "unlearning" a mathematical specification language like Z in favour of spending six months becoming a moderately skilled user of an awkward teletype language.

This description of PVS might seem to be negative, but many interactive theorem provers fit this description. For many theorem provers, the user interface is not as important as the logical engine that does the reasoning. As a result, users of theorem provers are often confronted with a system-specific specification language, usually one-dimensional and based on ASCII. Since mathematics uses special symbols and operators, a translation is needed from the mathematically oriented language to the specification language, which reduces the readability and can introduce errors. If the specification becomes unreadable, the user might prefer the blackboard to do the calculations and consider using the theorem prover to check it afterwards.

Some constructors of theorem provers have recognized that the interface should be improved, as discussed during the UITP (User Interfaces for Theorem Provers) workshops [7, 4]. To improve the readability, the interface should use mathematical notations, as used, for example, during lectures on the blackboard. A good example of such an improved interface is the Jape system [20], where the user works with a two-dimensional and familiar notation. The next step is to integrate the theorems, proofs and documentation into one single document, as in Mathematica [22] and Maple[16].

Now that we have successfully achieved our own initial goals, in the form of a stable, well-tested (mathematical-)document preparation system, the time is ripe to couple it to other tools, such as program verifiers. This document describes how we have combined Math/pad with PVS. We discuss the potential benefits of the combination seen from the viewpoint of someone wanting to use formal methods for the construction of computer programs, and we discuss the architecture of the combined system for the benefit of anyone wanting to investigate combining the Math/pad system with other programming tools. The system we have implemented runs under Unix and may be downloaded from http://www.win.tue.nl/cs/wp/mathspad.

2 User Model

The recent Ph.D. thesis by Matteo Vaccari [21] is illustrative of what we ultimately want to achieve. In his thesis, Vaccari discusses the calculational construction of hardware circuits, where the first 6 chapters contain theoretical discussions of relation algebra, circuits and regular language recognizers, while the later chapters contain simulations of the circuits using Tangram [18] and a machine verification of the theory using PVS [14]. Vaccari used Math∫pad in the process of developing and documenting the "theoretical" designs in the initial chapters, and then hand-coded these into the forms acceptable to Tangram and PVS. (See figure 1.)

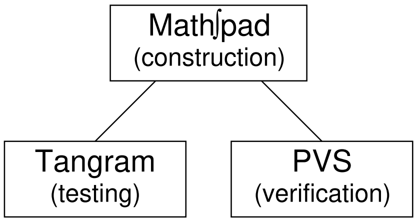

Fig. 1. The User Model

The use of *two* additional and entirely independent systems to check the "theoretical" designs gives a remarkable level of confidence in the reliability of Vaccari's designs that could not have been achieved by using any one of the systems on its own. Tangram is a system comprising a language, a simulator and a compiler developed at Philips Research Laboratories, Eindhoven, for the design of asynchronous hardware circuits. Using it, Vaccari was able to *test* that his designs functioned according to specification. In addition, Tangram has features to analyse the efficiency of a circuit design (including area, speed and energy consumption), and warns against unimplementable features. The PVS system comprises its own, quite different, specification language based on higher-order logic. Using it, Vaccari was able to formally *verify* all the lemmas and theorems leading up to and including the final circuit design. Vaccari comments in his thesis that neither systems showed up any errors in the calculated designs; however, the use of Tangram gave practical feedback, whereas the use of PVS obliged him to clarify certain elements in his calculations.

Independent checks are crucial to improving confidence, but there is one major weakness in the procedure adopted by Vaccari: namely, the lack of any formal link between the mathematical language in which his designs were constructed, the language of Tangram in which his designs were tested, and the language of PVS in which his designs were verified. This, however, is a weakness shared by all validation systems (theorem provers, model checkers, etc.) that we know of since such systems are invariably based on a language that is different to the actual implementation language used by "real" programmers. Practical reality compounds the problems drastically: since systems are subject to continual modification and evolution, it is almost inevitably the case that what is verified (or even tested) is not what is executed.

We believe that the use of a system like Math∫pad can make a substantial contribution to overcoming this weakness. Math∫pad is a structure editor — the user of Math∫pad manipulates, in fact, an abstract structure which is viewed on-screen as a mathematical expression but which can also be viewed as a Tangram program or as a collection of theorems and proofs in the PVS system.

Of course, matters are not quite as simple as we have just sketched. The process of viewing an abstract structure on screen involves, by design, a very simple transformation of the structure into display events, whereas the process of transforming the structure into a Tangram program is much less simple, and the process of converting it into a collection of PVS theorems and proofs —the topic of this paper— is far from trivial. But this is essentially what Vaccari did in his thesis, mostly by hand but also with the aid of a number of automated tools. Our goal in developing the interface with the PVS system was to automate this process as much as possible.

A tool like Math∫pad has the potential to be useful as an interface for several backend engines, such as symbolic computation systems and theorem provers. Many of those systems have a teletype interface and the mathematical content is often difficult to read and written in an unfamiliar syntax. Furthermore, each system uses its own syntax, which makes it virtually impossible to switch from one system to another. In Math∫pad, the user works with the familiar syntax, while the generated output is less important. As it is possible to generate output in another markup language, it is not too difficult to generate the input that is needed for a particular backend engine. For a normal user, Math∫pad should hide all the knowledge that is needed to use the backend engine and translate the input and output of the backend engine to the syntax familiar to the user. For an expert user, the connection with the backend engine should be easy to construct and maintain.

Since there are many possible backend engines with their own markup languages, the connection between Math∫pad and a backend engine should be as generic as possible. A connection with one particular backend engine would not be too difficult to construct and Math∫pad could be tuned for that backend engine. However, if a connection with a different backend engine is needed, the same work has to be done all over again. Therefore, the core Math∫pad system does not contain specific knowledge about one particular backend, but it provides the functionality to add that knowledge.

In the following sections, the connection we have made between Math∫pad and the PVS system is described. The PVS system was chosen because the Math∫pad documents with human readable proofs created by Vaccari and their PVS versions were available to us, thus providing a substantial test-base for our ideas. Furthermore, the PVS system is a non-trivial system and was likely to expose problems of a general nature when connecting Math∫pad to other systems.

3 An Example

A simple example will serve to illustrate the difference between mathematical calculation and PVS-style verification.

3.1 Mathematical Calculation

The example, in the popular Feijen style of proof presentation[11], in figure 2 is taken from Vaccari's thesis[21].

A law about *map* and *fold* is the following: given R and S such that

$$R \circ S{\times}S \quad = \quad S \circ R$$

then

$$fold_n.R \circ map_n.S \quad = \quad S \circ fold_n.R$$

The proof is by induction on n; for $n = 1$ it is trivially true. For $n + 1$ we have

$$\begin{array}{ll}
 & fold_{n+1}.R \circ map_{n+1}.S \\
= & \quad \{ \quad \text{definitions} \quad \} \\
 & R \ \circ \ \iota{\times}fold_n.R \ \circ \ S{\times}map_n.S \\
= & \quad \{ \quad \text{fusion} \quad \} \\
 & R \ \circ \ S{\times}(fold_n.R \circ map_n.S) \\
= & \quad \{ \quad \text{induction hypothesis} \quad \} \\
 & R \circ S{\times}(S \circ fold_n.R) \\
= & \quad \{ \quad \text{proviso: } R \circ S{\times}S \quad = \quad S \circ R\text{; fusion} \quad \} \\
 & S \ \circ \ R \ \circ \ \iota{\times}fold_n.R \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & S \circ fold_{n+1}.R
\end{array}$$

Fig. 2. The Formatted Example

In the example, a law is given about *map* and *fold*, together with a proof that the law is correct. The proof, although very simple, illustrates well the advantages of good, clear mathematical notation.

Consider the calculation introduced by the words "For $n+1$ we have". Note, first, the invisible use of the associativity of composition in the first two steps[1]. In the first step $fold_{n+1}.R$ is replaced by $R \ \circ \ \iota{\times}fold_n.R$, and $map_{n+1}.S$ is replaced by $S{\times}map_n.S$. The combined effect is to replace the top line in the calculation by

$$(R \ \circ \ \iota{\times}fold_n.R) \ \circ \ S{\times}map_n.S$$

where the parentheses indicate the grouping resulting from the two replacements. Note now that the second step groups the subterms differently. In the second step the subterms $\iota \times fold_n.R$ and $S \times map_n.S$ are "fused" together to form the

[1] Here multiplication has precedence over composition, denoted by a small circle. The meaning of the operators is not relevant to the current discussion.

subterm $S{\times}(S \circ fold_n.R \circ map_n.S)$. That is, the associativity of composition has been applied implicitly between the first and second steps transforming the expression displayed above to

$$R \circ (\iota{\times}fold_n.R \;\circ\; S{\times}map_n.S)$$

Such uses of associativity occur very frequently in calculations and, as here, a practised scientist would not make its use explicit. (In fact, another invisible step in the proof involves exploiting the fact that the symbol ι denotes the identity of composition.)

A second point to note about this proof is that "fusion" appears twice in the hints. Both hints refer to the same law, but the law is used in different directions in the two instances (once from left to right and once from right to left).

A final point about this little calculation is the non-explicit use of the transitivity of equality. What is proved is that the top line

$$fold_{n+1}.R \circ map_{n+1}.S$$

is equal to the bottom line

$$S \circ fold_{n+1}.R$$

but this is not stated explicitly since it is immediately clear from the structure and layout of the proof.

3.2 PVS Verification

Although the law and the proof are given in an informal manner, the manual translation to PVS is straightforward, as is shown in figure 3.

The translation may indeed be easy to carry out by hand, but the result is complex, is far from being readable and does not come anywhere near to the way that human beings wish to see proofs presented. The statement of the theorem is readable but this is misleading: the o and $*$ operators are overloaded. Since you can not define new binary operators in PVS and the number of operators that can be overloaded is small, it is very likely that the PVS specification will become unreadable, as binary operators have to be replaced by functions with two arguments. Furthermore, the precedence of the overloaded operators can not be changed, which leads to confusion if another precedence is assumed. In the example, the precedence of the o and $*$ operators is different in the PVS version, which decreases the readability.

Another factor that contributes to the unreadability is the requirement to be explicit about the use of the identity of composition. The line with the comment "add unit of composition", for example, involves a complex "path expression" indicating to which subterm the rule is applied, in a manner akin to the way that paths through a directory structure had to be typed in before the existence of pointing devices.

But most importantly, the proof script as shown in the example is but a very small part of what the user sees while the proof is being built. After each step in

The PVS definition:

```
fold_map:  THEOREM  R o (S*S) = S o R
              IMPLIES fold(n,R) o map(n,S) = S o fold(n,R)
```

The PVS proof script:

```
(induct "n" 1)                        %induction
1  (grind)                            %basis: trivial
   (rewrite "id0")
   (rewrite "id1")
2  (skolem!) (ground)                 %step
   (skolem!) (ground)
   (expand "fold" :if-simplifies t)   %definition
   (expand "map" :if-simplifies t)    %definition
   (assoc-rewrite "fusion" :dir RL)   %fusion lemma
   (inst?) (ground)                   %induction hypothesis
   (replace*)
   (rewrite "id1")                    %remove unit of composition
   (rewrite "id0" 1 ("R" "S!1") 1 RL) %add unit of composition
   (assoc-rewrite "fusion")           %fusion lemma
   (rewrite "id0")                    %remove unit of composition
   (replace*)                         %proviso
   (rewrite "comp_assoc")             %associativity of composition
```

Fig. 3. The PVS Example

the proof script, PVS will display the intermediate results and the current goal, which leads to several pages of formulae in the highly unreadable PVS-speak! A straightforward proof has thus been turned into an intellectual feat!

4 Building the Interface

4.1 Communication with PVS

From a user interface point of view, PVS is an extension of Emacs, which connects the proof engine to the Emacs interface and the Tcl toolkit. The user can edit files containing theorems and use the proof engine to construct the proof interactively. Since the proof engine is basically a lisp interpreter with state information, the user interface of the engine is hidden from the user by a collection of pull-down menus in Emacs. With these menus, the user can perform all the actions that might be needed to manipulate files, theorems, lemmas and proofs. However, to construct a PVS proof, the user has to enter plain lisp commands to apply tactics to a goal. To allow some proof planning, an interface with a Tcl program is available to keep track of the subgoals.

Since PVS is a closed system and cannot be modified, the available PVS interface had to be used. The first problem was the existing Emacs interface, which complicates the communication with PVS. Luckily, the PVS system consists of a core system connected to Emacs with a collection of Emacs lisp files, as shown in figure 4. Emacs contains a lisp interpreter which is used to load the lisp files for the PVS communication. These configuration files extend Emacs with new functions and menus to provide a PVS specific interface. With these additional functions, Emacs is able to communicate with the PVS core system, using its standard input and output.

By using the PVS core system directly, the communication is simplified and easier to maintain. However, the protocol used between Emacs and the core system is not documented, probably because the constructors of PVS didn't envisage a different interface to that core system. Therefore, the protocol had to be extracted from the lisp configuration files used by Emacs and the messages that are sent between Emacs and the core system. With some detective work in the form of a wrapper script that monitors these messages, we were able to reconstruct the most important parts of the protocol, which was sufficient to use the core system without the Emacs interface.

The PVS core system is a lisp interpreter and receives lisp commands as input, which can be used to update or inspect the state of the system or to prove a theorem. The output of the core system consists of a combination of commands to update the state of Emacs and the results of proving a theorem. The core system might also construct temporary files and instruct Emacs to open them, which is mainly used for help files and the Tcl interface. The Emacs interface cleverly hides the lisp input with a collection of pull-down menus, while the mixed output is parsed and separated into several buffers. For the average user, only the buffer with the results of a proof are of interest.

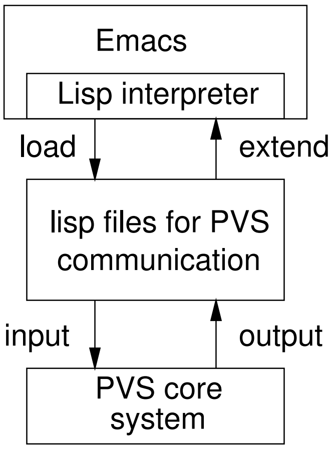

Fig. 4. The PVS structure

The PVS core system operates in three modes: a mode for managing the state, a mode for making the proofs and a debug mode. Since the active mode affects the commands that Emacs has to send, the system uses synchronization points when it switches to a different mode and notifies Emacs. The debug mode is only used if the system receives incorrect input, and this mode is ended by resetting PVS.

To construct a different user interface for the PVS core system, the user interface had to simulate the actions performed by Emacs, such that the core system could not notice the difference. As our plan was to hide PVS as much as possible from the user, only a subset of the actions available in Emacs were made available in the new interface.

4.2 The Math/pad Infrastructure

The PVS interface has been constructed as a loadable module. For this purpose, Math/pad provides an interpreted language which can be used to extend the interface and to load modules. It is also possible for a loadable module to extend the language with new functions, variables and types. The infrastructure of the entire system is shown in figure 5.

The interpreter can be used to customise Math/pad to a particular need. With the interpreted language, the user can define new functions to combine common sequences into a single function. These functions can be used in pop-up menus and keyboard definitions to customise the interface and the keyboard usage.

For each extension, Math/pad will load an interface definition file to adjust the menus and keyboard definitions. Depending on the complexity of the extension, the interface definition file can include a dynamic library, which can extend the interpreted language with new functions, types and variables. With these new

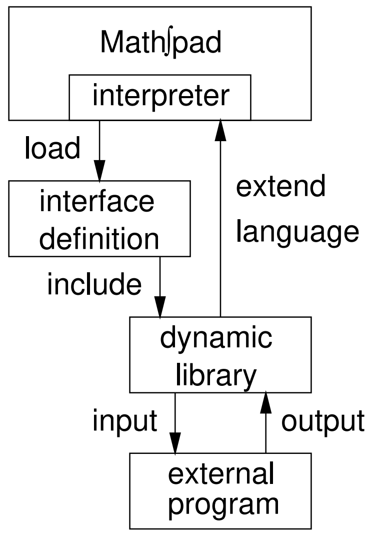

Fig. 5. The Math∫pad infrastructure

language items, the user can extend the menus and keyboard definitions and further customise the extension.

The combination of the interface definition file and the dynamic library can communicate with the external program through the standard input and output of the program. In order to do that, input has to be generated in the correct syntax for the particular program and the output of the program has to be parsed. As the interpreted language is not yet suited to the complex task of parsing the output, a dynamic library is usually needed if the output has to be parsed. When the output does not need to be parsed, some preprocessing of the output can be performed by adding a filter to the external program.

The interpreted language (see figure 6 for an example) is an imperative language, based on the guarded command language. It supports sequential composition, selection and repetition, but not recursion. Procedures are defined with a prototype, which is used to pass the arguments correctly, that is, to dereference variables where needed. Procedures can have local variables with the normal scope rules. To support callback functions, one additional operator is added to support lazy evaluation, that is, to pass an argument to a function such that it will be evaluated by that function at the correct time, for example after a filename is selected instead of before the file selector is opened.

The language supports a standard set of operators which can be overloaded by defining functions for each combination of arguments. This enables an extension to define new types with sensible operators, without the need to reconstruct the parser for the interpreted language.

Since some extension might have special needs for the content of menus, the strings in the interpreted language are in the Unicode encoding. This ensures that almost any symbol that an extension might need will be available for the pop-up menus and messages. For mathematical or foreign extensions, this will increase the readability.

4.3 The PVS Interface Library

The PVS module consists of a dynamic library for communicating with the PVS core system and an interpreted file to adjust the interface with Math/pad. The purpose of the dynamic library is threefold. First, it interprets the Math/pad document to extract theorems and proofs. Second, it generates the input that is sent to the PVS core system. Third, it parses the output that is generated by the core system.

As a Math/pad document is structured, the generation of theorems from a proof given in the Feijen style is not so difficult. For the example in figure 2, each of the five steps has to be correct, so we can generate a theorem for each step. Since the syntactical differences between the PVS input and the Math/pad version are not very different, generating these theorems is straightforward, once the definitions of the templates are correct. Extracting the proof for these theorems is also possible, as the hint contains keywords that indicate which strategies are applicable. In the example, the keyword "fusion" indicates that the fusion lemma is used as a rewrite rule. The keyword "definition" indicates that some definition has to be expanded and the keyword "induction" indicates that a premise is used as a rewrite rule, where the premise can be constructed from the proof itself by using the first and last expressions. However, the hints are not always precise enough, as is indicated by the PVS version of the proof. The additional details are automatically applied by a human reader of the formatted proof, without complaining. The reader will apply the trivial laws, such as the "identity of composition" and "associativity of composition", when needed and the direction in which an equality law is used is determined by trial and error. A complex dialogue with the author could be used to get these additional details, but we decided to define additional PVS strategies which simulate the behaviour of a human reader:

- a strategy to apply a rewrite rule in both directions,
- a strategy to retry a given strategy after applying the trivial laws, if that strategy fails the first time,
- a strategy to apply a rewrite rule modulo composition.

These strategies have their limitations, as it is likely that rewrite rules are applied incorrectly. However, the theorems are usually small and their proofs are short, so it is less likely that something will go wrong. In the event that a theorem can not be proven, an indication that the given hint is not sufficient to prove that step should be a reasonable reply from the system, as a reader might have the same problems with it as PVS.

For the other part of the example, the extraction of the theorems would require a combination of natural language processing and logical reasoning, for which a general solution is difficult. Therefore, this part is still missing from the current interface.

Once the theorems and proofs are known, they have to be converted to the specification language used by PVS. Since the output generated by Math/pad depends on the templates that are used, it is possible to generate valid PVS input from Math/pad expressions without much additional programming. However, the expressions appear in a certain context and the identifiers should have a certain type, otherwise PVS will generate parse or type errors. Although the context and type information can be stored in the document as hidden information, we chose to use a default context, where certain definitions and identifiers are predefined. This approach is quite common in documents with many identifiers, as it releases the author from the burden of mentioning the type of an identifier over and over again.

In order to give feedback to the user, the PVS output should be parsed and converted to familiar syntax. If a theorem is correct, all is well and a simple message should be sufficient. Otherwise, a warning or error message should be generated, indicating the problem and possibly a solution. If PVS does not need the generated proof completely, it might be that the given hint is incorrect or over-complete and Math/pad will suggest to adjust the hint in order to avoid confusing the reader. If PVS is unable to prove the theorem, the hint might be incomplete or an error might have occurred. By inspecting the output and comparing the expressions Math/pad could suggest that an identifier is incorrect or that a particular law might be applied. Since the user is not familiar with the PVS language, the output of PVS should be parsed and shown in the language as used in the document with a familiar syntax. However, the expressions in Math/pad are constructed with templates, which are used to generate the PVS expression. Therefore, these templates should also be used to parse the PVS output, which is complicated by the possible ambiguities in the definition of these templates. At the moment, this part is still missing from the experimental interface.

The PVS output also contains commands which are handled by Emacs. For each command, the PVS module will either ignore it or translate it to the new interface. For example, after the PVS core system has finished a proof, it will tell Emacs to open a buffer with the PVS file that contains the proven theorem. In Math/pad, that PVS file is generated by a step in a proof and of no interest to the user, so Math/pad will highlight the step that generated the PVS file.

The library adds the functions `pvs_check_hint` (to check the selected hint), `pvs_start` (to start PVS) and `pvs_add_keyword` (to define a keyword like "induction" mentioned above and the related PVS strategies). These functions, together with the already available functions, are used in the pop-up menus to extend the interface of Math/pad, for example to start PVS and to check a selected hint. The library also adds the variables `pvs_initialized` and `pvs_in_checker`, which can be used to inspect the status of PVS, and `pvs_context_dir`, `pvs_hint_file` and `pvs_lemma_name`, which are used to customise the generation of PVS files.

Since Math/pad uses Unicode internally, the strings that are part of the library, such as error messages, have to be converted to Unicode before they are used. This conversion uses a translation table to check whether the string has been customised by the user. This leaves a library with an additional method of customisation: by converting a string with the translation table, it can be adjusted by the user. In the PVS module, the string "PVS_HEADER" is used as the header of the PVS file, which defines the context of the generated theorem. By defining a translation for this string, the correct header is used.

In addition to the theory-specific keywords, there are four keywords with a special meaning. Each of these keywords is used in a special case:

- INITSTEP is used to initialise the PVS proof and to remove universal quantifiers.
- FINISHSTEP is used to finalise the PVS proof by applying all the trivial steps,
- EXPRESSIONSTEP is used when an expression occurs within a hint. Expressions in hints are regarded as assumptions and will result in a premise.
- STOPPVSPROOF is used when the proof fails and PVS has to leave the proof mode.

Without these four keywords, a correct proof script can not be constructed. Therefore, the interface definition file has to define these keywords with the `pvs_add_keyword` function.

The dynamic library that is used to communicate with PVS is written in C and consists of about 1000 lines of code. 35 percent is used to separate the PVS output and to handle the lisp requests from PVS, 15 percent is used to parse the PVS proof output and 20 percent is used to extract theorem and proof from the selected hint.

4.4 The Definition File

The PVS dynamic library handles the communication with the PVS core system and provides the interpreted language with a collection of high-level functions. With these functions, the pop-up menus of Math/pad have been extended with PVS specific commands or submenus. The interpreted language is also used to initialize and customize the PVS library, for example by filling the keyword list and setting up the context. Some parts of the definition file for PVS are shown in figure 6.

First, the dynamic library is included, meaning that the functions and variables from that library become available to the interpreter. It is also possible to include other definition files, which can be used to divide the different aspects of the interface over separate files.

After the dynamic library has been included, the function `pvs_reset` is defined, which is used to reset PVS if something goes wrong. This function could

```
Include "libpvs.so"

Function pvs_reset()
{
    if (pvs_initialized) {
        send_signal(2, "PVS Session");
        send_string(":reset\n", "PVS Session");
        pvs_in_checker := 0;
    }
}
Menu PVSMathSpad {
    Options Pin;
    Title "PVS Link";
    "Start"         : pvs_start("PVS Session");
    "Check Hint"    : pvs_check_hint(1);
    "Reset"         : pvs_reset();
    "Exit"          : send_string("(pvs::lisp (ILISP:ilisp-restore))
                         (pvs-errors (exit-pvs))\n", "PVS Session");
}
Menu  Misc {
    "PVS"   : PVSMathSpad;
}
Keyboard Global {
    `M-p` `s` : pvs_start("PVS Session");
    `M-p` `c` : pvs_check_hint(1);
    `M-p` `r` : pvs_reset();
}
Translation English {
  "PVS-shell"           : "PVS Session";
  "PVS_HEADER"          : " [t: TYPE+] : THEORY
 BEGIN
 IMPORTING tuples[t]
  n,m: VAR upfrom(1)
  R,S,T,U: VAR rel
";
}

pvs_context_dir := "/home/river/pvs-test";
pvs_hint_file := "hint";
pvs_lemma_name := "hintlemma";
pvs_add_keyword("STOPPVSPROOF", "(quit)\nY\n\"nil\"\nno\n",0);
...
pvs_add_keyword("induction",
                "(then* (inst?)(ground)
                  (try-triv-step (bidi-replace*)))\n", 1);
```

Fig. 6. The Interface Definition File

also be part of the dynamic library, but defining it in the interface definition file is more flexible, as it can be adjusted more easily.

Once all the functions are available, they can be linked to a pop-up menu and the keyboard. The interface definition language has special constructions to make this as easy as possible. A pop-up menu is defined by making a list of menu items, each containing a description and either the function to be called or the submenu to be opened. In the example, the menu called `PVSMathSpad` gives the user access to four PVS-specific functions. The menu itself is added as a submenu to the menu called `Misc`, which lists miscellaneous features.

Three functions are made available through keyboard shortcuts. After the `Meta-p` prefix, the key `s` will start PVS, the key `c` will check the selected hint and the key `r` will reset PVS.

To customise the PVS library, two translation strings are defined. As explained earlier, a translation for the string "PVS_HEADER" is given to set the context for the generated theorems. In general, this translation mechanism is used to customise the messages from Math/pad, as these are all in English and perhaps not clear enough (as in 'folder' versus 'directory').

At the end, the variables are initialised and the database of keywords is filled. At this point, the definition file is used as a script file to execute the functions while the definition file is loaded, which is used to futher customise the library.

5 Related Work

There are already some projects to improve the interface of PVS. TAME [2] is a layer on top of PVS for reasoning about timed automata and consists of a number of strategies to reduce the number of steps made in a typical PVS proof to the number of steps made in a hand-made proof. With these additional strategies, the user of TAME will not be exposed to the low-level steps and commands needed in PVS, thereby making the commands field specific. However, since the PVS interface is used, there is still a gap between the notational conventions used by PVS and those used in the documentation.

The system PAMELA [9] is designed to check partial correctness of VDM-like specifications in the area of code generators. By providing a connection with PVS, the system supports a larger class of specifications, using PVS to discharge proof obligations. The connection between PAMELA and PVS is made by extending PVS with additional commands and adding a Tcl/Tk interface which communicates with the Emacs system. Although this approach works, the modifications to PVS indicate that using a different theorem or a different interface would also require such changes. Furthermore, as the existing Emacs interfaces is still used, it does not remove the burden of using multiple interfaces and multiple specification languages.

Merriam constructed the PVS proof command prompter[13], which extends PVS with an additional input method for the proof commands to improve the PVS interface and to decrease the cognitive overhead for the user. The prompter uses a fill-in form to ask the user for the arguments that might be used for a given command.

GrammaTech and Formal Systems Design & Development are working on an environment for integrating formal methods tools to improve industrial acceptance of formal methods[1]. The environment will use active documents with embedded objects, with CORBA to handle the object distribution. The use of embedded object might cause some problems with the writability of the documents. That approach is also used by FrameMaker and Word, which are not the best word processors for mathematically oriented documents, as they have problems with context switches and treat mathematical expressions as images.

Simons has been working on a system to combine proofs in Isabelle[15] with documentation[19]. The system uses the structured documentation technique introduced by Knuth[12] to allow one file to contain both the proofs and the documentation and uses programs to separate those. This solves the problem of combining several files into one document, at the expense of using different languages in a single file, namely, LaTeX for formatting the document, Isabelle for specifying the proof and the meta language to instruct the programs. For a user, this mix of languages might be confusing.

The ILF system [10] offers a uniform interface for several automated theorem provers and it removes the burden of translating the specification files to the languages used by these theorem provers. The ILF system does not require any changes to the existing theorem provers and works like a server, which sends proof obligations to the available theorem provers and handles the results. Although ILF hides the specific languages and options of the theorem provers, it does add its own specification language, based on PROLOG.

6 Conclusions

The goal we set ourselves in this project was to automatically translate mathematical calculations to PVS proofs. An automatic translation is of course much more difficult than one done by hand. Nevertheless the goal was feasible, given that Vaccari had written his thesis with the Math∫pad system so that all the documents needed to test the connection between PVS and Math∫pad were already available to us. The goal has been achieved except for the interpretation of natural language linking together different calculations. There are also still some problems hiding the PVS language from the user.

Math∫pad does not help the user to construct the PVS files which are needed to get started. Therefore, the connection only works if there are already some PVS files with the required definitions. These files must be constructed by someone who is conversant with both Math∫pad and PVS. However, only a limited number of such experts are needed; (ultimately) other users can exploit the benefits of formal verification with the PVS system without a six-month training period.

The connection has been made without adjusting PVS in any way. That is, the same version of PVS can be used with the Emacs interface and the Math∫pad interface. Although the Emacs interface had to be separated from the PVS core system, this process is not very difficult and can easily be repeated for the next version of PVS, assuming that the internal interface does not change drastically. The conversion from version 2.1 to version 2.2 of PVS was a matter of updating the initialization file for the PVS core system, which can be constructed by monitoring the communication between Emacs and the PVS core system.

In order to build a different user interface for an existing theorem prover, the theorem prover should have a clearly separated user interface and core system. For PVS, this structure is not directly visible, but after a closer look, the separation is not very difficult, although the documentation is missing.

The use of loadable modules in the form of dynamic libraries is a powerful technique and allows easy extension of a system, as is shown by applications like Netscape, the Linux kernel, the GIMP and Photoshop. It allows modules from different sources to combine their strength in order to improve the total system. If theorem provers were available as modules, the main system could choose the best module for a given job. With some effort, it is possible to use an existing theorem prover as a module. However, every theorem prover uses its own input format, output format and user interface, which makes it very difficult to combine the power of multiple theorem provers for a single project. Perhaps the MathML or OpenMath languages will be useful in this respect.

PVS seems to be at the correct level of automation for our purpose. An automatic theorem prover could not verify whether the hints are meaningful and would require additional testing. A low-level theorem prover would need additional information to finish the proof or high-level tactics have to be introduced.

References

1. P. Anderson, M. Goldsmith, B. Scattergood, and T. Teitelbaum. An environment for intergrating formal methods tools. In Bertot [7]. See also: `http://www.grammatech.com/papers/uitp.html`.
2. Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In Backhouse [4], pages 147–156. See also: `http://www.win.tue.nl/cs/ipa/uitp/papers/Archer.ps.gz`.
3. R.C. Backhouse. Archive of the mathematics of program construction group. Available online at `http://www.win.tue.nl/cs/wp/papers/`, July 1998.
4. R.C. Backhouse, editor. *Workshop on User Interfaces for Theorem Provers*, Computing Science Reports, July 1998. International Workshop, see also `http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.
5. R.C. Backhouse, R. Verhoeven, and O. Weber. Math∫pad: A system for on-line preparation of mathematical documents. *Software – Concepts and Tools*, 18:80–89, 1997. See also: `http://www.win.tue.nl/cs/wp/mathspad/`.
6. Roland Backhouse and Richard Verhoeven. *Math∫pad Ergonomic Document Preparation*, version 0.60 edition, February 1996. Manual of the Math∫pad system. See also: `http://www.win.tue.nl/cs/wp/mathspad/`.
7. Yves Bertot, editor. *Workshop on User Interfaces for Theorem Provers*, September 1997. International Workshop, see also `http://www-sop.inria.fr/croap/events/uitp97-papers.html`.

8. Eerke Boiten, John Derrick, Howard Bowman, and Maarten Steen. Consistency and refinement for partial specification in z. In Marie-Claude Gaudel and James Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 287–306. Springer, 1996.
9. Bettina Buth. *Operation Refinement Proofs for VDM-like Specifications.* PhD thesis, Institute of Computer Science and Practical Mathematics of the Christian-Albrechts-University Kiel, February 1995. See also: `http://www.informatik.uni-bremen.de/~bb`.
10. Ingo Dahn. Using ILF as an interface to many theorem provers. In Backhouse [4], pages 75–86. See also: `http://www.win.tue.nl/cs/ipa/uitp/papers/Dahn.ps.gz`.
11. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, Berlin, 1990.
12. D.E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.
13. N.A. Merriam and M.D. Harrison. What is wrong with GUIs for theorem provers. In Bertot [7]. See also: `http://www.cs.york.ac.uk/~nam/uitp97.ps.gz`.
14. S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. See also: `http://pvs.csl.sri.com/`.
15. Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover.* Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.
16. Darren Redfern. *The Maple Handbook.* Springer, 1996.
17. John Rushby. What is pvs? Available online at `http://pvs.csl.sri.com/whatispvs.html`, November 1998. Contains a description of PVS.
18. Frits D. Schalij. Tangram manual. Technical Report UR 008/93, Philips Electronics N.V., 1996.
19. Martin Simons. Proof presentation for Isabelle. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 259–274, Murray Hill, NJ, August 1997. Springer-Verlag.
20. Bernard Sufrin and Richard Bornat. User interfaces for generic proof assistants part II: Displaying proofs. In Backhouse [4], pages 147–156. See also: `http://www.win.tue.nl/cs/ipa/uitp/papers/Sufrin.ps.gz`.
21. Matteo Vaccari. *Calculational Derivation of Circuits.* PhD thesis, Dipartimento di Informatica, Università degli Studi di Milano, May 1998. See also: `http://dotto.usr.dsi.unimi.it/~matteo/tesi.ps.gz`.
22. Stephen Wolfram. *The Mathematica Book.* Cambridge University Press, third edition edition, 1996.