[12] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Some lessons learned. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 482–500, Odense, Denmark, April 1993. Volume 670 of *Lecture Notes in Computer Science*, Springer-Verlag.

[13] David Lorge Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[14] David Lorge Parnas. Some theorems we should prove. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, pages 155–162, Vancouver, Canada, August 1993. Number 780 in Lecture Notes in Computer Science, Springer-Verlag.

[15] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[16] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[17] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[18] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[19] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction*, volume 121 and volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, Holland, 1988. In two volumes.

References

[1] Michael J. Beeson. *Foundations of Constructive Mathematics*. Ergebnisse der Mathematik und ihrer Grenzgebiete; 3. Folge · Band 6. Springer-Verlag, 1985.

[2] Michael J. Beeson. Proving programs and programming proofs. In *International Congress on Logic, Methodology and Philosophy of Science VII*, pages 51–82, Amsterdam, 1986. North-Holland. Proceedings of a meeting held at Salzburg, Austria, in July, 1983.

[3] Michael J. Beeson. Towards a computation system based on set theory. *Theoretical Computer Science*, 60:297–340, 1988.

[4] Ermanno Bencivenga. Free logics. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic–Volume III: Alternatives to Classical Logic*, volume 166 of *Synthese Library*, chapter III.6, pages 373–426. D. Reidel Publishing Company, Dordrecht, Holland, 1985.

[5] William M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, September 1990.

[6] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[7] Raymond D. Gumb. *Programming Logics: An Introduction to Verification and Semantics*. John Wiley and Sons, New York, NY, 1989.

[8] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. Foundations of Computing. MIT Press, Cambridge, MA, 1988.

[9] C. L. Heitmeyer and B. G. Labaw. Consistency checks for SCR-style requirements specifications. Technical report, US Naval Research Laboratory, Washington DC, August 1993. In press.

[10] C. A. Middelburg and G. R. Renardel de Lavalette. LPF and MPL_ω—a logical comparison of VDM SL and COLD-K. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, pages 279–308, Noordwijkerhout, The Netherlands, October 1991. Volume 551 of *Lecture Notes in Computer Science*, Springer-Verlag. Volume 1: Conference Contributions.

[11] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga, NY, June 1992. Volume 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

that the automated reasoning required to solve these examples is really quite elementary. For example, the truth of many of the theorems is independent of the interpretation of the functions and predicates that appear within them (the examples of Figures 4, 5, 11 and 13 are like this), and they can therefore be proved by essentially propositional reasoning. Simple automated reasoning techniques are used effectively in a tool described by Heitmeyer and Labaw [9]; the tool discovered a number of significant errors in a substantial set of tables that had already been subjected to extensive human review. Clearly, there is no point in using a full theorem prover when simple techniques suffice. However, if simple techniques prove inadequate on some applications (and Heitmeyer and Labaw report that their tool did generate some "false positives"), then we suggest that employing a system such as PVS may prove more economical in the long run than stretching simple techniques beyond their limits. And, of course, if we want to establish application-level properties of the specification (such as that it achieves some intended function), rather than simply establish local forms of consistency for a tabular representation, then a full theorem prover designed for verification-type problems is likely to be essential.

Finally, although the purpose of this note has been to demonstrate theorem proving, we hope the reader may also find something of value and interest in style of specification supported by PVS. In particular, we believe that predicate and dependent subtyping (and the attendant obligation to prove theorems in order to decide type-correctness) provide very effective solutions to issues (such as partial functions) that pose considerable difficulties in other specification languages with mechanized support. Also, although we can appreciate some of the attractions of the tabular approach to specification, we suggest that those who practice this approach should also consider whether direct expression in a traditional logic such as that of PVS might not be advantageous in some circumstances. Tables are an effective way to present control-dominated constructions, but simple functions and algorithms may be more perspicuous when presented in traditional logic.

Acknowledgments

We have had stimulating discussions with David Parnas on the content of this note. We are grateful to Jeff Joyce and to Paul Miner for bringing Parnas' paper to our attention, and for the support and encouragement of Connie Heitmeyer of the US Naval Research Laboratory.

```
floor 1   1 {\lfloor #1 \rfloor}
/     key 1 {\div}
```

The result of LaTeX-printing the definition of `palindrome?` is shown below.

palindrome?$(l,\ (n \mid\ l\ +\ n\ \leq\ N\ +\ 1),\ A)$: bool
$=\ (\forall\ (i:\ \text{nat}):\ i\ <\ \lfloor n \div 2 \rfloor\ \Rightarrow\ A(l\ +\ i)\ =\ A(l\ +\ n\ -\ 1\ -\ i))$

3 Conclusions

We appreciate this opportunity to demonstrate the effectiveness of a modern theorem prover on problems representative of those that arise in the software engineering and verification methodology developed by David Parnas. All but one of the theorems posed by Parnas were proved automatically by the tcc strategy of PVS. The entire development (transcription of the theorems into PVS and their proof) took about an hour—essentially the time required to interpret the theorems and type them in. We speculate that proof of somewhat harder theorems could also be largely automated: it has been our experience that any given development tends to generate many similar theorems. An expert could therefore develop an appropriate PVS strategy for the class of theorems generated by a particular development. However, we do not consider it productive to attempt the construction of automatic proof procedures for genuinely hard theorems—our experience is that it is better and faster for a skilled and knowledgeable user to guide the proof than to attempt heroic automation.

One issue not raised by the theorems examined here is what to do when an automated proof attempt fails. This can happen for two reasons: the theorem may be true but the automated procedures are inadequate to prove it, or the theorem may be false. In our experience, it requires some skill to distinguish between these cases, and it may not be easy to develop that skill when relying on automated proof procedures.

Parnas [14] writes that the theorems considered here are "more difficult than the majority of the theorems that arose in the documentation and inspection of the Darlington Nuclear Plant Shutdown Systems," and that that activity "resulted in about 40 kg. of such trivial tables" (as those that generated the theorems considered here, not including the palindrome example). He continues "if these theorems can be proven automatically by today's theorem proving programs, we should be using those programs." We hope to have demonstrated that those with problems similar to those described by Parnas should consider using modern theorem proving systems such as PVS.

Some may argue that using a full verification system such as PVS to prove these simple theorems is like using a sledgehammer to crack a walnut, and will observe

```
palindrome?(l, (n | l+n <= N+1), A): bool =
   (FORALL (i: nat): i < floor(n/2) => A(l+i) = A(l+n-1-i))
```

Notice that the type of the second argument is now restricted to the range `1 ...N+1-l`. With this adjustment, the tcc becomes

```
palindrome?_TCC1:
  OBLIGATION (FORALL (i: nat), l, (n: posnat | l + n <= N + 1):
     i < floor(n / 2) IMPLIES l + i > 0 AND l + i <= N)
```

which is (easily) provable. With more precise type constraints on the predicate `palindrome?`, we need to adjust the statements of `fig13` and `fig15` so that they remain type-correct:

```
fig13: THEOREM (EXISTS l, (n|l+n<=N+1): palindrome?(l, n, A))
    => nonempty?({l | (EXISTS (n|l+n<=N+1): palindrome?(l, n, A))})

fig15: THEOREM nonempty?({n|(EXISTS (l|l+n<=N+1): palindrome?(l, n, A))})
```

This correction to the specification has no effect on the proofs of these theorems (it just makes their statements valid in the logic of PVS).

It might seem that Parnas' partial term logic has avoided this complication involving dependent types. In one sense it has: his specification has a meaning even if accesses can occur outside the domain of the array. However, when we come to prove a theorem such as `fig15` it is not enough to know that the expression `palindrome?(l, n, A)` has *some* meaning—we need to evaluate its *actual* meaning in order to decide the truth of the theorem. That will require evaluating the expression `A(l+i) = A(l+n-1-i)` for all combinations of the variables concerned. In order to consider the truth of this equality, we need to know whether the expressions on either side are defined or not. Thus, in formally evaluating this expression as it appears in `fig15`, a theorem prover based on a partial term logic would probably pose definedness lemmas identical to the tcc shown earlier. Whereas a mechanization of a partial term logic would encounter such definedness obligations (possibly several times) at proof time, PVS encounters them at typecheck time. This provides earlier error detection, requires each definedness obligation to be discharged only once, and allows more information to be provided in the types.[6]

There is one other feature of PVS we should mention before closing: PVS provides a LaTeX-printer that can be customized by simple tables to recreate the preferred notation of a particular application area. The following two-line incantation causes it to employ the symbols used by Parnas for the floor and division functions.

[6] It would be interesting to try this in a system such as IMPS that uses a partial term logic.

The definition is given in terms of Hilbert's ε (choice) operator and is followed by a lemma which states that the `floor` of a real number `x` is the largest integer less than or equal to `x`. The definition and lemma come from the PVS prelude; we do not explain the definition or the proof of the lemma here, since their level of difficulty is out of keeping with the rest of this note; the general PVS user needs only to understand the property stated in `floorprop`.

The tcc-strategy built-in to PVS is unable to prove the theorem `fig15` automatically. It is necessary for the user to supply some "insight": namely, that the reason this theorem is true is that palindromes of length 1 occur at every position. Thus, to finish the proof after the tcc-strategy has done its work, it is necessary for the user to suggest two instantiations (one for the length `n` and another for the starting position `l`) and to invoke `floorprop`.[4]

Although it looks as though we have completed the assignment suggested by Parnas, a few loose ends remain. Typechecking a PVS specification can lead to the generation of proof obligations called tccs that must be discharged before the specification is considered type-correct. In the present case, most of the tccs are trivial[5] and are discharged automatically by the tcc-strategy that also disposes of the main theorems suggested by Parnas. However, the declaration of `palindrome?` generates some tccs that inspection shows to be unprovable (in fact, false)—for example:

```
palindrome?_TCC1:
  OBLIGATION (FORALL (i: nat), l, n:
     i < floor(n / 2) IMPLIES l + i > 0 AND l + i <= N)
```

The issue here is that the definition of `palindrome?` requires accessing the values of the array `A` at index positions between `l+i` and `l+n-1-i` where `l` is known to be of type `index`, `n` is a `posnat`, and `0 <= i < floor(n/2)`. Because PVS is a logic of total functions, it requires us to prove that all accesses to `A` will be within its domain—this is the content of the tcc `palindrome_TCC1` shown above. Unfortunately, it is easy to see that some of these accesses could actually be outside the domain of `A` and that the tcc is consequently false. The way to repair this deficiency is to realize that once we have chosen a value for `l` (the starting position of the putative palindrome), the length of the longest palindrome that can lie within the array is restricted to `N+1-l`. Thus not all combinations of `l` and `n` are valid arguments to `palindrome?` and we need to restrict its domain appropriately. This is done by means of a *dependent type* declaration—a type that depends on the *value* of some term appearing earlier in the specification. Here the appropriate specification is the following.

[4]Notice that this theorem remains true (and becomes stronger) if the existential quantifier is replaced by a universal one. This variant also has an easier proof, since the starting position `l` is skolemized and we do not need to supply an instantiation for it.

[5]For example, the declaration of the type `index` requires that we show $0 < 1$ and $1 \leq N$.

```
A: VAR [index -> T]
n: VAR posnat
l: VAR index
palindrome?(l, n, A): bool
fig13: THEOREM (EXISTS l: palindrome?(l, n, A))
                    => nonempty?({l | palindrome?(l, n, A)})
```

Notice that we have given a signature, but no interpretation for the predicate `palindrome?`. This is because the truth of the theorem `fig13` is independent of the interpretation of this predicate (the theorem is essentially identical to that of `Fig11`). The theorem is proved in exactly the same way as the previous two.

2.9 The Theorem of Figure 15

This theorem was originally posed to us as

$$n > 1 \Rightarrow \textbf{nonempty}(\{n | \exists l, (\forall i, 0 \leq i < \lfloor n \div 2 \rfloor \Rightarrow A[l+i] = A[l+n-1-i])\}).$$

This formula contains both a free occurrence of n and a bound occurrence and we could make no sense of the free occurrence. The label given to the theorem by Parnas suggested that it was intended to assert that the set of lengths of palindromes in the nonempty array A is nonempty. This indicates that the antecedent to the implication should be $N > 0$ rather than $n > 1$.

With this interpretation, we can state the theorem in PVS as

```
fig15: THEOREM nonempty?({n | (EXISTS l: palindrome?(l, n, A))})
```

We have confirmed with David Parnas that this is the intended form and that the original was a typographical error. Notice that we do not need to explicitly impose the condition `N > 0` because this is embedded in the type (`posnat`) specified for `N`.

In order to evaluate this theorem, we do need to give an interpretation to the predicate `palindrome?`:

```
palindrome?(l, n, A): bool =
       (FORALL (i: nat): i < floor(n/2) => A(l+i) = A(l+n-1-i))
```

This in turn requires a definition for the function `floor`:

```
floor(q: real): int = epsilon({n: int | n <= q
                            AND (FORALL (m: int): m <= q => m <= n)})
floorprop: LEMMA floor(x) <= x
                  AND (FORALL (m: int): m <= x IMPLIES m <= floor(x))
```

2.7 The Theorem of Figure 11

Here we are required to prove

$$(\exists i, B[i] = x) \Rightarrow \textbf{nonempty}(\{j' | B[j'] = x\}).$$

The PVS version is an almost exact transliteration:

```
fig11: THEOREM (EXISTS (i: index): B(i) = x)
                    => nonempty?({j: index | B(j) = x})
```

Sets are identified with predicates in PVS,[3] and the set-valued expression `{j: index | B(j) = x}` is simply a syntactic variation on the predicate definition `(LAMBDA (j: index): B(j)=x)`. The function `nonempty?` is from the PVS prelude (i.e., built-in) theory called `sets`, which supplies definitions for various set-theoretic constructs. We excerpt the ones relevant here:

```
sets [T: TYPE] : THEORY
 BEGIN
  member(x:T, a:set): bool = a(x)
  empty?(a:set): bool = (FORALL (x:T) : NOT member(x,a))
  nonempty?(a:set): bool = NOT empty?(a)
 END sets
```

The proof of `fig11` is obtained by automatically expanding the definitions of `nonempty?`, `empty?`, and `member`, followed by skolemization, propositional simplification, and heuristic instantiation.

2.8 The Theorem of Figure 13

Here we are required to prove

$$(\exists l, (\forall i, 0 \leq i < \lfloor n \div 2 \rfloor \Rightarrow A[l+i] = A[l+n-1-i])) \Rightarrow$$
$$\textbf{nonempty}(\{l' | (\forall i, 0 \leq i < \lfloor n \div 2 \rfloor \Rightarrow A[l'+i] = A[l'+n-1-l])\})$$

The idea is that the array A (indexed by $1 \ldots N$) has a palindrome of length n starting at position l if $A[l+i] = A[l+n-1-i]$ for all i from 0 up to (but not including) $\lfloor n \div 2 \rfloor$. The theorem states that if there exists such an l, then the set of such ls is not empty.

We could transcribe this directly as written into PVS, but to save some typing and to provide a clearer specification, we prefer to define an auxiliary predicate: `palindrome?(l, n, A)` will be true when there is a palindrome starting at position `l`, of length `n` in the array `A`. Then we have:

[3] PVS is a simply-typed higher-order logic, so the axiom of comprehension (identifying sets with predicates) is sound.

2.5 The Theorem of Figure 7

Here we are required to prove, for any array B indexed by the integers $1 \ldots N$,

$$(\exists i, B[i] = x) \vee (\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x)).$$

N is presumably some fixed constant, but it is not clear whether B is intended to be an arbitrary constant or a variable. Since skolemization will reduce the latter to the former, it doesn't matter which we choose here, so we have used a constant. Parnas treats B as a partial function with domain $1 \ldots N$, whose value is not defined for other indices. In PVS, we define $1 \ldots N$ as a subtype (called `index`) of the integers and introduce B as a *total* function on this domain. The range type of B is not specified by Parnas, so we introduce an uninterpreted type T to serve this purpose.

```
N: posnat
index: TYPE = {i: int | 1<= i & i <= N} CONTAINING 1
T: TYPE
B: [index -> T]
```

We can then state the required theorem as:

```
x: VAR T
fig7: THEOREM (EXISTS (i: index): B(i) = x)
           OR (FORALL (i: index): B(i) /= x)
```

Notice that we have overloaded `x` to be both a variable of type `T` and one of type `real` (defined earlier). The PVS typechecker disambiguates these names in context by virtue of their types. Because we specify the quantified variables `i` to be of type `index`, we do not need to explicitly mention the range qualification used in Parnas' form of the theorem (it is embedded in the definition of the `index` type).

The proof of this theorem follows by skolemization, propositional simplification, and heuristic instantiation.

2.6 The Theorem of Figure 8

This is a trivial variant on the previous example (it follows by De Morgan's rule).

$$\neg((\exists i, B[i] = x) \wedge (\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x))).$$

The PVS version is obvious

```
fig8: THEOREM NOT ((EXISTS (i: index): B(i) = x)
                AND (FORALL (i: index): B(i) /= x))
```

and its proof is the same as the previous one.

We do not discuss the proofs of Figures 9 and 10 since they are merely simplified instances of the previous two.

```
nonneg_real?(x: real): bool = (x >= 0)
nonneg_real: TYPE = (nonneg_real?) CONTAINING 0
sqrt: [nonneg_real -> nonneg_real] % (could return a pair if required)
```

The clause `CONTAINING 0` is provided in order to discharge the tcc that will require the nonemptiness of this subtype to be demonstrated.

PVS does not provide `domain` or `intype?` constructions (though it would be easy to incorporate them), so we have to interpret the theorem stated by Parnas as follows:

```
x: VAR real
fig4: THEOREM x < 0 => nonneg_real?(-x)
```

Unlike the earlier theorems that used local bindings for `x` and explicit quantification, here we have used a global declaration and implicit quantification (all formulas in PVS are automatically closed by universally quantifying their free variables). This is purely a syntactic convenience and has no semantic consequences. The proof follows by skolemizing, expanding the definition of `nonneg_real`, and ground arithmetic reasoning.

Notice that another way to generate the theorem of interest would be to propose any expression involving `sqrt(-x)` in a context where `x` is known to be negative. For example:

```
variant: THEOREM x<0 => sqrt(-x) = sqrt(-x)
```

The typechecker will then automatically generate a theorem identical to `fig4` above as a tcc necessary to ensure type-correctness of the expression `sqrt(-x)`.

2.4 The Theorem of Figure 5

This is a trivial variation on the previous example:

$$x > 0 \Rightarrow \mathbf{domain}(\sqrt{x}).$$

The PVS specification is

```
fig5: THEOREM x > 0 => nonneg_real?(x)
```

and the proof proceeds just as before.

2.2 The Theorem of Figure 3

Here the requirement is to prove

$$(\forall x, \neg((x < 0 \wedge x = 0) \vee (x < 0 \wedge x > 0) \vee (x > 0 \wedge x = 0))).$$

In PVS, this becomes

```
fig3: THEOREM
    FORALL (x: real): NOT ((x < 0 AND x = 0)
                           OR (x < 0 AND x > 0)
                           OR (x > 0 AND x = 0))
```

and again the proof follows directly from skolemization and ground arithmetic.

2.3 The Theorem of Figure 4

Here we are required to prove

$$x < 0 \Rightarrow \mathbf{domain}(\sqrt{-x}).$$

Parnas employs a partial term logic [13] in order to accommodate partial functions such as square root,[1] whereas PVS uses classical logic and total functions. However, PVS provides predicate and dependent types that can be used to constrain the domains of what would otherwise be partial functions. Thus, for example, division is a total function on the domain that excludes 0 in its second argument position.

Here, we define the predicate `nonneg_real?` to be true of just the non-negative reals. Given a predicate `p` in PVS, the parenthesized construction `(p)` is a type-expression that denotes the subtype of the domain of `p` that satisfies `p`. Thus, in particular, `(nonneg_real?)` denotes the subtype of the reals consisting of just the non-negative reals. We then introduce `nonneg_real` as a synonym for this subtype. Next, we specify the square root function `sqrt` to be a function that takes a `nonneg_real` as its argument and returns one as its value.[2] We do not supply an interpretation for the `sqrt` function since none is needed for the proofs considered here.

[1] The logic used is similar to that of Beeson [2] (more accessible references to Beeson's work are [1, Chapter 6, Section 1] and [3, Section 5]). For a general account of "free logics" (of which Beeson's is an example) see [4], and for a brief discussion of their application to partial term logics see [19, volume I, chapter 2, section 2]. PX [8] is a computational logic based on these ideas, while a (higher-order) logic of this kind is mechanized in the IMPS system [5, 6], and variants have been proposed for other specification languages [10]. Gumb [7, Chapter 5] uses a free logic to express facts about execution-time errors in programs.

[2] Alternatively, by a trivial modification, we could specify it to return a pair consisting of a `nonneg_real` and a `nonpos_real`.

PVS proof steps can be composed into larger steps that we call "strategies" (these are akin to the "tacticals" of LCF-like systems) that can deal automatically with various classes of elementary theorems. One such strategy is the "tcc-strategy" built-in to PVS for the purpose of automatically discharging many of the "type-correctness conditions" (tcc's) that arise during typechecking. This strategy skolemizes, iteratively expands explicit definitions, performs simple heuristic instantiation, and then applies decision procedures for propositional calculus and ground (i.e., variable free) arithmetic. It is able to prove all but one of the theorems posed by Parnas. The one that it fails to prove requires something close to "insight" and can be completed with just a couple of additional user-supplied steps.

Implicit in Parnas' "challenge" to prove these theorems automatically is the expectation that the theorem and supporting specification text submitted to the prover should be in a form close to that employed by Parnas. In the following section, we describe the specification text submitted to PVS and also explain the theorem proving mechanisms used to prove each theorem.

2 The Theorems and Their Proofs

The theorems suggested by Parnas appear as figures in [14]; not all the figures appearing there are theorems, however—some are specifications (expressed as tables) whose various well-formedness constraints give rise to the theorems. In this section we describe each of the theorems in a separate subsection, labeling each with the figure number from [14] of the corresponding theorem.

2.1 The Theorem of Figure 2

For real number x, prove that

$$(\forall x, (x < 0 \vee x = 0 \vee x > 0)).$$

In PVS this is represented as

```
fig2: THEOREM FORALL (x: real): x < 0 OR x = 0 OR x > 0
```

The proof follows by skolemization and ground arithmetic.

The PVS prover includes decision procedures for linear arithmetic over both integers and reals [15–18]. That means it can decide *any* true ground formula of arithmetic involving the propositional connectives, equality over uninterpreted functions symbols, the arithmetic relations $<, >, \leq, \geq$, and the operators of addition, subtraction, multiplication and division, but with the latter two restricted to the linear case (i.e., one argument must be a literal constant). Theorem provers that lack automation of arithmetic can be tedious to use, since even ostensibly nonnumerical specifications often generate numerous proof obligations concerning (usually trivial) arithmetic facts.

Using PVS to Prove Some Theorems of David Parnas*

Reprint from *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, Vancouver, Canada, August, 1993.
Springer-Verlag LNCS 780, pp. 163–173.

John Rushby and Mandayam Srivas
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

David Parnas [14] describes some theorems representative of those encountered in support of certification of software for the Darlington nuclear reactor. We describe the verification of these theorems using PVS.

1 Introduction

David Parnas [14] describes some theorems representative of those encountered in support of certification of software for the Darlington nuclear reactor. After noting that these illustrative theorems "will appear trivial to mathematicians," Parnas [14] writes "if these theorems can be proven automatically by today's theorem proving programs, we should be using those programs. If these theorems still require human intervention, perhaps the developers of theorem proving programs would like to turn their attention to this type of theorem." In this note, we describe the transcription of these theorems into the language of PVS and their mechanized verification using the PVS proof checker [11].

PVS is not an automatic theorem prover; it is primarily intended for the proof of hard theorems such as those that arise in verification of interesting algorithms [12]. Consequently, PVS provides rather powerful automation of the lower levels of deduction (for example, decision procedures for ground arithmetic), but is generally guided by a human operator. We find this arrangement to be the most productive for kinds of theorems that arise in the problem domains we have studied. However,

*This work was supported by the US Naval Research Laboratory contract N00015-92-C-2177