Automatic Verification of Pascal Programs

DISSERTATION

submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of Doctor of Mathematics

presented by

EDOUARD MARMIER

Dipl. Math. University of Zurich
born September 12, 1945
Citizen of Zurich and Sevaz (Canton of Fribourg)

Accepted on the recommendation of

Prof. Dr. N. Wirth
Prof. Dr. E. Engeler

Juris Druck + Verlag Zurich
1975
5. DISCUSSION AND CONCLUSIONS

In this chapter we comment on properties of PASCAL which were of importance to the PPPV project. Then some remarks evaluating the PPPV are made. The chapter ends with a personal view on program verification and automatic theorem proving.

5.1. PASCAL

PASCAL is the subject language and the implementation language of the PPPV. We consider both aspects in turn.

5.1.1. PASCAL as the Subject Language

No field related to the Art of Programming depends as much as program verification on the simplicity of programming languages. Without going into details, the author states that PASCAL was an obvious and agreeable choice as a subject language. The concepts of PASCAL are few in number and clearly defined.

Program verification, in the absolute sense of the word, is the art of proving programs to be correct in every respect, and one important demand on programming languages is that they should facilitate such proofs as far as possible. Apart from pointers and real numbers, the one and only trouble spot the author has found in PASCAL is common to every other programming language he knows: one cannot prove with reasonable effort that no reference is made, in a program, to an uninitialized variable or component of a variable. The axiomatization of PASCAL [HoW I] explicitly states that if a variable x is declared of type T, then it may be assumed in any proof relating to the scope of x that the value of x is an element of T. But no implementation of the full language guarantees this, and it is difficult to conceive of one which would. Moreover, such
an automatic initialization by implementation is not a desirable thing at all: it makes the discovery of initialization errors less likely. A better scheme is to let implementations initialize variables to some value 'undefined' and to abort the execution of programs which perform calculations with this value. (For entire variables, the PPPV uses a similar concept.) Thinking of packed structures, however, one foresees severe complications here too.

Habermann, in [Hab 1], rides a violent attack on PASCAL. Lecarme and Desjardins give a defence in [LeD 1]. Subrange types are one particular topic of their debate, and it is interesting to note that as far as program verification is concerned, subrange types present no substantial problem. It suffices, whenever a variable $x$ of subrange type $m..n$ is assigned a new value, to prove that this value lies between $m$ and $n$, and whenever an assumption is made about $x$, then $m \leq x \leq n$ must also be assumed. (This scheme generalizes readily to programming languages which allow any "invariant" to be imposed on a variable, not just relations $m \leq x \leq n$.)

Constant declarations are the element of the PPPV's subject language the author feels most uneasy about. Consider a sorting program the first few lines of which are

```plaintext
const n = 100 {number of elements to sort};
type Index = 1..n;
var A : array [Index] of ... {elements to sort};
```

If the PPPV verifies this program, then we have a program which correctly sorts $n = 100$ elements. We know nothing about the otherwise identical programs one obtains by setting $n = 101$, or $n = 100000$, or $n = 1$. To enable verification for arbitrary constants $n$, the necessary extensions to the assertion language are straightforward: a formula describing the relevant properties of $n$ must be provided by the programmer, e.g.

```plaintext
const n = 100 { $n \geq 1.$};
```

The verification system would check that the value specified for the constant indeed satisfies the formula, but would process the rest of the program with respect to the formula only, independently
of the value. Severe problems in the compilation phase are foreseen: it must be shown that all declarations and statements which refer to declared constants conform to the rules of PASCAL for arbitrary values of the constants. E.g., it must be shown that logn ≥ 1 in

\[
\begin{align*}
\text{const } n &= 100 \{ .n \geq 2.\};
\log n &= 7 \{ .2**(\log n-1) \leq n \leq 2**\log n.\};
\text{var } B : \text{array } [1..\log n] \text{ of } ...
\end{align*}
\]

5.1.2. PASCAL as the Implementation Language

It is with a deep sigh of envy that the author read page V-3 of [Deu 1]. In the implementation of Deutsch's system, it is assured that there occurs in storage at any moment at most one copy of each expression or subexpression. His system (p. V-3)

"... implements this with a function (HCONS c1 c2) which constructs a new node with the given components if none exists, or returns the existing node if there is one."

Let us call this implementation strategy "maximal sharing", as opposed to "minimal sharing" in which every expression in storage is referenced at most once. An implementor of a verification system has to decide whether to adopt maximal sharing, minimal sharing, or an intermediate strategy.

For theorem-proving applications, the author believes maximal sharing to be best. In terms of PASCAL, if two pointers p1 and p2 respectively reference expressions E1 and E2, maximal sharing allows one to decide whether E1 and E2 are identical simply by asking whether p1 = p2. The economy of storage space achieved by it is of paramount importance. Information related to an expression (such as the set of variables occurring in it) can be affixed to the expression, without necessitating a copy of the information being made together with every further copy of the expression. In some automatic theorem provers, the negation of a formula is processed almost as often as the formula itself, and with maximal sharing it is not prohibitive to construct and retain the negation together with the formula. Just the fact that every occurrence of an expression in storage is unique constitutes a simple and useful
principle of high value in the programming of a theorem prover.

However, the PPPV is based on minimal sharing. Minimal sharing leads to fairly rapid storage overflow, entails a large amount of copying time, and in many detail respects leads to an undesirable conceptual complexity (perhaps because one is forced to "optimize" in various places). The reason why it was chosen is the absence of a garbage collector in the PASCAL 6000 system.

The following considerations may seem out of place to LISP programmers, but since we are discussing the first theorem prover written in PASCAL, they may be of interest to PASCAL programmers. It is certain that they have heavily influenced the PPPV.

5.1.2.1. garbage collection?

 Powerful automatic theorem provers have the tendency to produce large amounts of formulas of only temporary use (or of no use at all), and it is indispensable that the storage space occupied by obsolete formulas be made available for re-use. Without garbage collection, a theorem prover would die from self-suffocation almost instantly.

We shall now argue that it is impossible to provide a safe garbage collector for the PASCAL run-time support system. Lists in PASCAL consist of nodes, which normally are (packed) records with variants, and pointers, which represent "arrows" between the nodes. We shall state our point in terms of a simple, although typical, example. A graph structure containing two kinds of abstract nodes, namely a "unary" kind with exactly one successor, and a "binary" one with exactly two, could have the following PASCAL representation:

type nodekind = (unary,binary);
nodeptr = †node;
node = record
case nk : nodekind of
unary : (info1 : ...; onlysucc : nodeptr);
binary: (info2 : ...; leftsucc, rightsucc : nodeptr)
end;

var p1,p2 : nodeptr
info1 and info2, which are not necessarily of the same type, denote specific information associated with each node, but are irrelevant to the structure of the graph. Now let us assume that a set of concrete nodes has been created in store:

We also assume that we have a garbage collector and that p1 and p2 are the only variables of type nodeptr which are not fields in a node. Then all nodes which do not lie on a path starting at the nodes referenced by p1 and p2 are inaccessible and will play no role in subsequent computations. Such nodes are garbage, and the garbage collector detects this by marking all nodes accessible from p1 and p2. Now in PASCAL, it is possible to convert a node of one variant to a node of another variant, simply by assigning a new value to a tag field. E.g. if p2↑ is a binary node in our example, the assignment

\[ p2↑.nk := \text{unary} \]

converts it to a unary one. (The PPPV executes such dangerous conversions in some critical sections, and totally overwrites nodes in other ones.) Therefore the garbage collector cannot a priori know whether a node it encounters is a unary or a binary one: to decide whether a node has one or two successors, it must rely on the value of the tag field of the node. Now if, as a result of a programming error, a tag field is incorrectly set, the garbage collector would perhaps follow nonexistent links in the marking phase, or, worse, fail to follow existing ones (e.g. if a node is mistakenly indicated as unary). A small error in a program may thus cause any garbage collector to foul the entire storage space, perhaps without giving any indication as to what went wrong. We conclude that a safe garbage collector cannot exist for PASCAL. (To make matters yet more difficult, PASCAL now allows records with
variants but with no tag fields: in this case, garbage collection is impossible).

Another problem is that all pointers would have to be initialized to nil when they come into existence. Pointers can be components of highly complex variables (file of record of array of ... of pointer), and initializations to nil become difficult.

If PASCAL provides no garbage collection, then storage space management is up to the programmer. One maintains a list of available space and uses a procedure, say with name Return, which inserts a node referenced by a parameter into the list of available space. Instead of a tacit garbage collector one has a possibly large number of calls of Return scattered throughout the program. Worse is the fact that if one forgets to call Return at some place, say inside a repetitive statement, some garbage will for ever occupy precious storage space.

Nevertheless, this approach had to be adopted for the PPPV. Now comes the final decision: should one allow sharing, i.e. more than one reference to one and the same node?. In the affirmative, every node must also have a reference counter associated with it, i.e. a variable or component whose value is the number of references to the node. Every statement of the form

\[ p.t\text{.suc}: = q \]  \hspace{1cm} \text{(1)}

entails one, say, of the form

\[ q.t\text{.reference\_counter} : = q.t\text{.reference\_counter} + 1 \]  \hspace{1cm} \text{(2)}

and immediately this introduces a dangerous source of errors: an omission of (2) at a certain point can cause disaster and may be extremely difficult to locate. (Note that in PASCAL, the actions (1) and (2) cannot be combined into a procedure, because fields of packed records cannot be var parameters.)

Fearful of the threats inherent in sharing, the author selected the minimal sharing policy for the PPPV. To state the case again: maximal sharing would have been preferred, but was rejected because
of apprehensions due to the inevitable absence of a garbage collector in PASCAL. It appears that the more complexity one's application has, the simpler should the elements implementing it be. At the outset, the author did not anticipate that the PASCAL list structure mechanisms would turn out to be a major source of troubles when used in a theorem-proving program.

5.2. Evaluation and Conclusions

At this point in a thesis or report on a software product, it is customary to point out the overwhelming merits of the product. One argues that in some sense it is superior to all systems comparable to it. For the sake of objectivity, some—not too many—of its weaknesses are acknowledged. One ends by asserting that the product represents a significant step towards the goal one is ultimately aiming at.

We shall follow this scheme to some extent only; in particular, the conclusions will not be favourable.

When evaluating the performance of the PPPV, one should take the following points into account:
(i) The PPPV is a non-interactive program.
(ii) It was built from scratch in the sense that the wide, albeit young, traditions of Artificial Intelligence programs written in LISP for a PDP-10 computer was of little help.
(iii) The PPPV program was written by one man alone.
(iv) No garbage collector was available in the implementation language.
(v) As expounded in the Introduction, the aims and attitudes associated with the PPPV project as a whole were not primarily of the Artificial Intelligence type.

These points make it rather difficult for any person to compare the PPPV with other systems, and the author, in particular, has never had the opportunity to work with another theorem prover or program verifier. The following views necessarily are subjective.
The most outstanding novel aspects of the PPPV are:

1) It has shown that solid language design and implementation principles can be applied to assertion languages as well as to programming languages. For instance, declaration mechanisms are accepted in modern programming languages as one feature which enhances clarity, readability, and safety; such mechanisms have been incorporated into the PPPV's assertion language. During the "compilation" phase of the PPPV, attributes of declared annotational variables, functions, predicates and lemmas are effective and the use made of annotational objects is checked for consistency with these attributes.

2) The PPPV rejects all input which does not conform to the rules of PASCAL. In particular, the PPPV guarantees that no preconditions of inference rules are violated. In this respect, other verification systems with ALGOL-like or PASCAL-like subject languages probably are unsound: a grave reproach to a program verifier. (An easy escape from this, of course, is simply to leave out procedures or to exclude global variables completely and to call this progress.)

3) Except for termination concerns, undefined array components and undefined or out-of-bounds var parameters, the PPPV proves that all expressions it encounters are meaningful. E.g. errors such as uninitialized entire variables, illegal array subscripts or division by zero are caught.

4) The context/knowledge stack mechanism and the departure from traditional flowchart-oriented ways of looking at programs enable a more natural verification process. Verification is guided by the structure of the input program in a manner clearly recognizable to a programmer using the PPPV.

Little or no attention is payed to these vital aspects in other program verification systems. Therefore the PPPV represents a significant step towards the practical attainment of verified software.

The PPPV is itself unsound in one respect: in its implementation, no heed is given to integer overflow. Because the CDC 6000 computer
gives no indication whatsoever in the case of integer overflow conditions, incorrect deductions are possible. The author's viewpoint, however, is that it cannot be the concern of software to compensate for irresponsible hardware.

The absence of notations for sets and multisets are felt to be the most important weakness of the PPPV's assertion language. Many assertions and invariants are most elegantly formulated in terms of sets and multisets, and they make shorter proofs possible in some cases. As a striking example, compare with what follows the troublesome formalisms we set up in section 2.2.2.3., or which [Suz 1] introduces, in order to formulate the permutation property for arrays:

For arrays B we define $M(B)$ to be the multiset consisting of all the values of all the elements of B. Let A and AO be two arrays such that, at some point in a program, $M(A) = M(AO)$ (i.e. A is a permutation of AO). We show that the interchange


leaves this property invariant. After "$x := A[i]$" we still have $M(A) = M(AO)$; using the priming mechanism to introduce new names, "$A[i] := A[j]$" replaces one array element by another and leads to

$$M(A) = M(A') - \{A'[i]\} + \{A'[j]\}$$

(+ and - denote the operations of inserting/deleting occurrences of elements in a multiset); "$A[j] := x"", with $x = A'[i]$, leaves us with

$$M(A) = M(A'') - \{A''[j]\} + \{A'[i]\}$$

The most difficult step is to note that $A''[j] = A'[j]$, independently of the values of i and j. Using this we obtain

$$M(A) = M(A') - \{A'[i]\} + \{A'[j]\} - \{A''[j]\} + \{A'[i]\}$$

$$= M(A')$$

$$= M(AO)$$

and the permutation property is preserved. Compared with this, the
Theorem proving convulsions of the PPPV in Appendix E are ridiculous. Being such powerful concepts, however, sets and multisets have powerful inference rules associated with them. In the light of section 5.2.1. below, the author expects them to lead to insurmountable problems for a non-interactive theorem prover.

Considering the limitations imposed by (i) - (v) at the beginning of this section, the author feels that the theorem-proving performance of the PPPV is quite respectable. Essentially all of King's programs have been verified (including the legality of array indices), and other more important and complex programs have successfully been processed, such as g.c.d. with multipliers, binary search, binary approximation to a monotonic function, several sorting algorithms including an unguarded version of Quicksort, and a program schema which stores all solutions of an integer problem (see Appendix E).

To repeat it a last time: the PPPV is a non-interactive program written essentially from scratch by one man alone in a language and system which provides no garbage collector. The PPPV attempts to prove more about programs than any other existing verification system, and a substantial amount of effort was spent on important aspects of program verification which are ignored by other authors. With this in mind, the author considers the PPPV as highly successful.

5.2.1. Contre Automatic Theorem Proving

One biting definition of Artificial Intelligence characterizes it by saying that it encompasses those computer-based activities in which one does not really know what one actually is doing. The author cannot say how much truth there is in this characterization in general, but does believe that it applies to the (Automatic) Theorem Proving subfield of Artificial Intelligence. Looking back at the evolution of the PPPV's theorem proving routines, he cannot honestly maintain that it did not apply to his own work.

The author furthermore believes that due to the incorporation of a theorem prover, the approach taken by the PPPV and similar systems presents many interesting facets, that in an intuitive sense it
perhaps even gives rewarding feelings to an implementor, but that it is basically wrong.

In what follows, the words "theorem", "proof", "theorem prover", etc., are understood with respect to an undecidable domain. The word "theorem prover" at first is employed in a strict sense: a theorem prover is a procedure which attempts to find a proof of a given formula and thereby develops some activity independent from the user, without being instructed what to do next after each step.

Problems with theorem provers originate in that we do not know what we want them to be capable of. For instance, it is assumed as self-evident that the more powerful a theorem prover is, the better. While nothing can be said against this if one's objective is to implement a theorem prover for its own sake, we shall argue that if a theorem prover is intended to serve as a helpful and, above all, reliable component of a larger system of practical value, then "power" is detrimental. A powerful theorem prover is one capable of carrying out many deductions. However, every deduction rule implemented in a theorem prover is likely to be also applied when it leads astray. Experience in fact shows that theorem provers, when applied unbridled to a given theorem they cannot prove immediately, produce floods of useless deductions generating useless formulas which in turn participate in yet more useless deductions. (Does this statement possess a theoretical confirmation? I.e., can it be or has it been shown that for every complete proof procedure for number theory, there is a class of theorems \( T[1], T[2], ... \) with shortest proofs respectively of lengths \( L[1], L[2], ... \) such that the proof procedure takes arbitrarily long, or at least time \( O(2^{*L[1]}) \), to prove \( T[i] \)?)

If a powerful theorem prover documents its activity by displaying all formulas it deduces, then its output generally cannot with reasonable effort be intellectually managed by a person. If less output is produced, it becomes difficult or impossible to see along which lines the proof of a non-trivial theorem is carried out.

And now, what if a theorem prover fails, after a certain time, to find a proof of a formula \( F \)? Does \( F \) have no proof, or is the theorem prover too weak? If the theorem prover produces much output, one can perhaps discover why it failed, although this task
is abominable; from little or no output one obtains little or no help.

We safely conclude that the activity of a powerful theorem prover quickly grows out of control in the case of difficult or unprovable formulas. As far as practical applications are concerned, therefore, we see that powerful theorem provers are useless. Weak theorem provers are those which generally fail on difficult propositions and perform unreliable on easy ones; they are also useless.

Must we therefore give up all hopes of formally checking by machine that programs are consistent with their assertions? Not necessarily, provided we discard the Artificial Intelligence approach. We should no longer attempt to simulate creative thinking on machines unfit for the task.

Most verification systems make interactive use of theorem provers, i.e. the activity of the theorem prover is guided by commands entered during the verification process. One wonders whether the name "theorem prover" remains adequate: wherein lies the difference between an interactively used theorem prover and a system for symbolic manipulation? When Knuth, in [Knu 1], reports that the MACSYMA system [MAC 1] was very useful, it remains that he, Knuth, and not the computer, was the theorem prover on that occasion. MACSYMA is but a tool, albeit a good one.

But it still is not clear whether the program verification process should necessarily be interactive. The style of interactive verification depends on the idiosyncrasies of the particular verifier used. If a program is verified at one interactive installation, how can the proof be communicated to a wider public? Man-machine dialogues are usually unfit for publication and, in the case of complex interactions, unlikely to be understood or even considered by many people. When it is the human who directs the theorem proving operations, then program verification reduces to little more than proof checking. If it is clear to a programmer how a program correctness proof shall be carried out in detail, then there seems to be no need for interaction except for struggling with a recalcitrant "theorem prover". If it is not clear to him, then he had better think again about the problem.
We must remember at this point that program verification as such is the act of checking whether a given program, given specifications, and given invariants or other proof suggestions are consistent. It certainly is not the art of discovering why a program works: the explanation of the basic ideas behind a program must be provided by its creator and in fact belongs to its documentation. When in quest for a solution to a problem, we may allow ourselves any imaginable aids, including, of course, interactive computing. Verification of a proposed solution is a different matter.

The author believes that when the activity on Automatic Theorem Proving (in the strict sense, at least), will have died out, we shall observe more work on the implementation of proof checkers than there is today. Efforts will be spent in the study of proof languages, i.e. of higher-level formal systems in which reasonings can be expressed in a manner both digestible for human readers and amenable to verification by a proof checker. Proof checkers will be used to scrutinize mathematical proofs, and perhaps program verification will profit from them. (There already are proof checking systems, e.g. the AUTOMATH program of de Bruijn et al. [Bru 1], [Bru 2]. The AUTOMATH language is one of low level.)

We emphasize that a proof checker is a decision procedure: for any proposed proof it is reliably decided in finite time whether or not the proof is flawless with respect to a given formal system. All practical programs the task of which is to check whether their input possesses a certain property are the most valuable if they are decision procedures. In particular, program verifiers, which process objects of high complexity, namely programs, should be kept sufficiently simple so as to be decision procedures.

Artificial Intelligence will perhaps prove successful in some areas; in program verification it is the wrong approach and doomed to failure.