NOTES ON THE DESIGN OF EUCLID

G. J. Popek
UCLA Computer Science Department
Los Angeles, California 90024

J. J. Horning
Computer Systems Research Group
University of Toronto
Toronto, Canada

B. W. Lampson, J. G. Mitchell
Xerox Palo Alto Research
Palo Alto, California 94304

R. L. London
USC Information Sciences Institute
Marina del Rey, California 90291

Euclid is a language for writing system programs that are to be verified. We believe that verification and reliability are closely related, because if it is hard to reason about programs using a language feature, it will be difficult to write programs that use it properly. This paper discusses a number of issues in the design of Euclid, including such topics as the scope of names, aliasing, modules, type-checking, and the confinement of machine dependencies; it gives some of the reasons for our expectation that programming in Euclid will be more reliable (and will produce more reliable programs) than programming in Pascal, on which Euclid is based.

Key Words and Phrases: Euclid, verification, systems programming language, reliability, Pascal, aliasing, data encapsulation, parameterized types, visibility of names, machine dependencies, legality assertions, storage allocation.

CR Categories: 4.12, 4.2, 4.34, 5.24

Introduction

Euclid is a programming language evolved from Pascal [Wirth 1971] by a series of changes intended to make it more suitable for verification and for system programming. We expect many of these changes to improve the reliability of the programming process, firstly by enlarging the class of errors that can be detected by the compiler, and secondly by making explicit in the program text more of the information needed for understanding and maintenance. In addition, we expect that effort expended in program verification will directly improve program reliability. Although Euclid is intended for a rather restricted class of applications, much of what we have done could surely be extended to languages designed for more general purposes.

Like all designs, Euclid represents a compromise among conflicting goals, reflecting the skills, knowledge, and tastes (i.e., prejudices) of its designers. Euclid was conceived as an attempt to integrate into a practical language the results of several recent developments in programming methodology and program verification. As Hoare [1973] has pointed out, it is considerably more difficult to design a good language than it is to select one's favorite set of good language features or to propose new ones. A language is more than the sum of its parts, and the interactions among its features are often more important than any feature considered separately. Thus this paper does not present many new language features. Rather, it discusses several aspects of our design that, taken together, should improve the reliability of programming in Euclid.

We believe that the goals of reliability, understandability, and verifiability are mutually reinforcing. We never consciously sacrificed one of these in Euclid to achieve another. We had a tangible measure only for the third (namely, our ability to write reasonable proof rules [London et al. 1977]), so we frequently used it as the touchstone for all three. Much of this paper is devoted to decisions motivated by the problems of verification.

Another important goal of Euclid, the construction of acceptably efficient system programs, did not seem attainable without some sacrifices in the preceeding three goals. Much of the language design effort was expended in finding ways to allow the precise control of machine resources that seemed to be necessary, while narrowly confining the attendant losses of reliability, understandability, and verifiability. These aspects of the language are discussed in more detail by [Barnard and Elliott 1977]. The focus here is on features that *contribute* to reliability.

Goals, History, And Relation To Pascal

The chairman originally charged the committee as follows: "Let me outline our charter as I understand it. We are being asked to make *minimal* changes and extensions to Pascal in order to make the resulting language one that would be suitable for systems programming while retaining those characteristics of the language that are attractive for good programming style and verification. Because it is highly desirable that the language and appropriate compilers be available in a short time, the language definition effort is to be quite limited: only a month or two in duration. Therefore, we should not attempt to design a significantly different language, for that, while highly desirable, is a research project in itself. Instead, we should aim at a 'good' result, rather than the superb." [Popek 1976] We defer to the Conclusions a discussion of our current feelings about these goals and how well we have met them.

The design of Euclid took place at four two-day meetings of the authors in 1976, supplemented by a great deal of individual effort and uncounted Arpanet messages. Almost all of the basic changes to Pascal were agreed upon during the first meeting; most of the effort since then has been devoted to smoothing out unanticipated interactions among the changes and to developing a suitable exposition of the language. Three versions of the Euclid Report have been widely circulated for comment and criticism; the most recent appeared in the February 1977 *Sigplan Notices* [Lampson et al. 1977]. Proof rules are currently being prepared for publication [London et al. 1977].

The System Development Corporation is currently implementing Euclid [Lauer 1977]. Since the implementation is incomplete and no sizable Euclid programs have been written, our expectations are still untested. Further experience may dictate changes in the language.

We developed Euclid by modifying Pascal only where we saw "sufficient reason." We see it as a (perhaps eccentric) step along one of the main lines of current programming language development: transferring more and more of the work of producing a correct program, and verifying that it is consistent with its specification, from the programmer and the verifier (human or mechanical) to the language and its compiler.

Our changes to Pascal generally took the form of restrictions, which allow stronger statements about the properties of programs to be based on the rather superficial, but quite reliable, analysis that the compiler can perform. In some cases, we introduced new constructions whose meaning could be explained by expanding them in terms of existing Pascal constructions. These were not merely "syntactic sugaring": we had to introduce them, rather than leaving the expansion to the programmer, because the expansion would have been forbidden by our restrictions. Because the new constructions were sufficiently restrictive in somé other way, breaking our own restrictions in these contolled ways did not break the protections they offered.

The main differences between Euclid and Pascal are

Visibility of names: Euclid provides explicit control over the visibility of names by requiring the program to list all the names imported into a routine (i.e., procedure or function) or module body, or exported from a module. The imported names must be accessible in every scope in which the routine or module name is used.

Variables: Euclid guarantees that two names in the same scope can never refer to the same or overlapping variables. There is a single, uniform mechanism for binding a name to a variable in a procedure call, on block entry (replacing the Pascal *with* statement), or in a variant record discrimination.

Pointers: The avoidance of overlapping is extended to pointers by allowing dynamic variables to be partitioned into *collections*, and guaranteeing that two pointers into different collections can never point to overlapping variables.

Storage allocation: The program can control the allocation of storage for dynamic variables explicitly, in a way that narrowly confines knowledge about the allocation scheme used and opportunities for making type errors. It is also possible to declare that the dynamic variables in a collection should be reference-counted and automatically deallocated when no pointers to them remain.

Types: Type declarations are generalized to allow formal parameters, so that arrays can have bounds that are fixed only when they are allocated, and variant records can be handled in a type-safe manner. Records are generalized to include constant components.

Modules: A new type-constructor, *module*, provides a mechanism for packaging a collection of logically related declarations (including variables, constants, routines, and types) together with initialization and finalization components that are executed whenever instances of the type are created or destroyed. This provides some of the advantages of abstract data types.

Constants: Euclid defines a *constant* to be a literal or a name whose value is fixed throughout the scope in which it is declared, but not necessarily at compile time. A constant whose value is fixed at compile time (as in Pascal) is called a *manifest* constant.

For statements: The parameter of the for statement is a controlled constant in Euclid. A module can be used as a *generator* to enumerate a sequence of values for the controlled constant.

Loopholes: Features of the underlying machine can be accessed, and the type-checking can be overridden, in a controlled way. Except for these explicit loopholes, Euclid is designed to be type-safe.

Assertions: The syntax allows assertions to be supplied at convenient points to assist in verification and to provide useful documentation. Some assertions can be compiled into run-time checks to assist in the debugging of programs whose verification is incomplete.

Deletions: Several Pascal features have been omitted from Euclid: input-output, real numbers, multi-dimensional arrays, labels and go to's, and functions and procedures as parameters.

The only new features which can make it hard to convert a Euclid program into a valid Pascal program by straightforward rewriting are parameterized type declarations, storage allocation, finalization, and some of the loopholes.

The balance of this paper presents the motivations and consequences of several of the changes.

Verification And Legality

One of our fundamental assumptions is that (in principle) all Euclid programs are to be verified before use. That is, we expect formal proofs of the consistency between programs and their specifications. These proofs may be either manual or automatic; we expect similar considerations to apply in either case. We used the axiomatic method of [Hoare and Wirth 1973] for guidance.

Perhaps the most obvious consequence of this assumption is the provision within the language of syntactic means for including specifications and intermediate assertions. Routines are specified by pre- and post-assertions; modules, by a pre-assertion, an invariant, an abstraction function, and specifications for exported routines and types. In addition, assertions may be placed at any point in the flow of control. (Most verifiers require at least one such assertion to "cut" each loop.) Effort invested in writing such assertions should pay off in more understandable, better-structured programs, even before the verification process is begun.

The basic assertion language consists of the Boolean expressions of Euclid. Most verifiers will require somewhat richer languages, containing, for example, quantifiers, ghost variables, and specification routines. Rather than picking a particular form for this extended language, we decided that extended assertions would be bracketted as comments; each verifier may choose a private syntax, without affecting Euclid compilers. (Indeed, a program might be augmented with two distinct sets of assertions, intended for different verifiers.)

Most programs presented to verifiers are actually wrong; considerable time can be wasted looking for proofs of incorrect programs before discovering that debugging is still needed. This problem can be reduced (although not eliminated) by judicious testing, which is generally the most efficient way to demonstrate the presence of bugs. To assist in the testing process, any scope in Euclid can be prefixed by *checked*, which will cause the compilation of run-time checks for all *basic assertions* (Boolean expressions not enclosed in comment brackets) within the scope; this includes all *legality assertions*, which will be discussed later. If any assertion

evaluates to False when it is reached in the program, execution will be aborted with a suitable message.

Because we expect all Euclid programs to be verified, we have not made special provisions for exception handling [Melliar-Smith and Randell 1977][McClaren 1977]. Run-time software errors should not occur in verified programs (correctness is a compile-time property), and we know of no efficient general mechanisms by which software can recover from unanticipated failures of current hardware. *Anticipated* conditions can be dealt with using the normal constructs of the language; most proposals for providing special mechanisms for exception handling would add considerable complexity to the language [Goodenough 1975].

We have also been led to a somewhat unorthodox position on uninitialized variables and dangling pointers. We do not forbid these syntactically (cf. [Dijkstra 1976] for a rather elaborate proposal), nor, for reasons of efficiency, do we supply a default initialization (e.g., to "undefined"). Our reasoning is as follows: verification generally places stronger constraints on variables (pointers) than that they merely have valid values when they are used--they must have *suitable* values. However, if a program can be verified without reference to the initial value of a variable (current variable to which a pointer points), then *any* value (variable) is acceptable.

Relying so heavily on verification has an obvious pitfall: suppose that the formal language definition and the implementation don't agree. (Indeed, for Pascal, they do not.) We could then be in the embarrassing situation of having failures in programs that have formally been proved "correct" [Gerhart and Yelowitz 1976]. Aside from some omissions and known technical difficulties (e.g., [Ashcroft 1976]), the major discrepancies between the Pascal definition and implementation take the form of restrictions imposed by the definition, but not enforced by the implementation. For example, "The axioms and rules of inference...explicitly forbid the presence of certain 'side-effects' in the evaluation of functions and execution of statements. Thus programs which invoke such side-effects are, from a formal point of view, undefined. The absence of such side-effects can in principle be checked by a textual (compile-time) scan of the program. However, it is not obligatory for a Pascal implementation to make such checks." [Hoare and Wirth 1973, p.337]

In the design of Euclid, we have made a major effort to ensure that there are no gaps between what is required by the definition and what must be enforced by any implementation (and that such enforcement is a reasonable task). Gaps have been eliminated by a variety of means: removing features from the language, extending the formal definition, placing more definite requirements on the implementation, and finally, introducing *legality assertions* as messages from the compiler to the verifier about necessary checking.

There are many language-imposed restrictions that must be satisfied by every legal Euclid program. In addition to syntactic constraints, many of them (e.g., declaration of identifiers before use) are easily checked by the compiler, and it would be silly to ask the verifier to duplicate this effort. Others (e.g., type constraints) can usually be checked rather easily by the compiler, but may occasionally depend on dynamically generated values. Still others (e.g., array indices within bounds, arithmetic overflow) will usually depend on dynamic information, although the compiler can often use declared ranges or flow analysis to do partial checking. (For example, $i := i + 1$ will obviously never assign a value that is too small if i was previously in range.) Our philosophy is that the verifier should rely as much as possible on the checking done by the compiler. In fact, unless the compiler indicates differently, the verifier is entitled to assume that the program is completely legal. The compiler is to augment the program with a *legality assertion* (which the verifier is to prove) whenever it has not fully checked that some constraint is satisfied. Any program whose legality assertions can all be verified is a legal program, with well-defined semantics.

The compiler may produce legality assertions only for certain conditions specifically indicated in the Euclid Report. They always take the form of Boolean expressions, and are usually quite simple (e.g., $i < 10$, $i = j$, p **not=** C.nil). Note that legality is a more fundamental property than correctness, since (a) it is defined as consistency with the language specification, rather than consistency with a particular program specification (a program could be consistent with one specification, and inconsistent with another), and (b) a program that is illegal has no defined meaning, and hence cannot be said to be consistent with any specification. Also note that a particular program is not sometimes legal and sometimes illegal (e.g., depending on whether $i = j$ on some run): the verifier must show that the legality assertions are *valid* (always true).

Later sections of this paper discuss some of the non-obvious legality conditions of Euclid.

Names And Scopes

In "Algol-like" languages the rules connecting names (identifiers) to what they denote (e.g., variables) give rise to some subtle, but troublesome, problems for both programmers and verifiers. Some variables, for example those passed as variable parameters, may be accessible by more than one name. Thus, assignment to x may change y: we will call this *aliasing*. Access to a global variable can accidentally be lost in a scope by the interposition of a new declaration involving the same name (the "hole in scope" problem). Conversely, failure to declare a variable locally may result in a more global access than was intended. (Such problems are generally not detected by compilers.) The intimate connection between a variable's lifetime and its scope frequently forces variables to be declared outside the local scopes in which they are intended to be used. Finally, the automatic importation of all names in outer scopes into contained scopes, unless redeclared, tends to create large name spaces with correspondingly large opportunities for error. For more complete discussions of these problems, and some suggested solutions, see [Wulf and Shaw 1973] and [Gannon and Horning 1975].

Several Euclid features are intended to remove these problems; they are discussed here and in the following two sections. Unlike the designers of Gypsy [Ambler et al. 1977], we did not discard the Algol notion of nested scopes, which seems to us to be a natural representation of hierarchy, and a good first approximation to the necessary name control. Rather, we have chosen to strengthen it by a number of restrictions.

The first restriction requires the programmer to control the "flow" of names between levels of abstraction by means of an *import list*. Every *closed scope* (routine or module body) must be accompanied by such a list specifying those names accessible in the containing scope that are to be accessible within the closed scope, and, in the case of variables, whether the access is to be read-only or read-write. Other names are simply inaccessible. An *open scope* (e.g., an Algol-like block) may access any name accessible within the scope that contains it.

The control supplied by import lists allows us to place a further restriction: no name accessible in a scope may be redeclared in that scope. Such a restriction would probably be intolerable in Pascal, where a scope has no "protection" against unwanted names from the outside, but it seems sensible in Euclid. In fact, it is generally a programming error to redeclare an imported name. Undiagnosed holes in scopes would certainly cause problems for the reader and maintainer, and for the human verifier.

In practice, we found it desirable to relax slightly the requirement of explicit importation. We do not wish every routine that uses built-in types, such as **integer**, or routines, such as abs(x), to import them explicitly. Many programs will have user-defined types and routines that are almost as widely used. Therefore, we have provided an overriding mechanism: constant, routine, and type names may be declared *pervasive* in a scope, which means that they will be implicitly imported into all contained scopes (and hence may not be redeclared). The standard Euclid types are all pervasive: therefore, a program cannot override them.

Euclid prohibits "sneak access" to variables by means of procedure calls. The name of a closed scope may not be imported (or used) if the names that are imported into its body are not also imported (accessible) at the point of use. It is this restriction that simplifies the enforcement of a complete ban on side-effects in functions (and hence in expressions). Functions cannot have variable parameters or import variables. Although they may import and call procedures, they cannot change any nonlocal variables by doing so: thus, they behave like mathematical functions. The possibility of side-effects in functions and expressions complicates the verifier's task, and we believe that their use is rather error-prone. We are willing to sacrifice a few well-known programming tricks that rely on "benign" side-effects in order to simplify life for the readers, maintainers, and verifiers of programs, and to open up new optimization possibilities for the implementors of the language. Programs involving functions with side-effects can be rewritten to use procedures instead.

Import lists are intended to make the interface to each closed scope explicit. However, the list supplied by the programmer is incomplete (for the reader) in two respects: 1) only names are given, not complete declarations, and 2) pervasive names do not appear. The compiler is expected to complete the interface description from its symbol table. It must augment the listing with information from the declarations of the imported names, and the user-defined pervasive declarations for that scope. Requiring the programmer to supply this information (which is mere duplication) would invite error, for no identifiable gain.

Aliasing And Collections

The disadvantages of aliasing (for programmers, readers, verifiers and implementors) have been well-documented [Hoare 1973, 1975] [Fischer and LeBlanc 1977]. If assignment to x has the "side-effect" of changing the value of y, it is likely to cause surprise and difficulty all around. However, programmers and language designers have been reluctant to eliminate all features that can give rise to aliasing, e.g., passing parameters by reference, and pointer variables. In designing Euclid, we took a slightly different approach: we kept the language features, but banned aliasing. Essentially, we examined each feature that could give rise to aliases, and imposed the minimum restrictions necessary to prevent them. Every variable starts with a single name: if no aliases can be created, then, by induction, aliasing will not occur.

The case of variable parameters to procedures is typical, and easily generalized to import lists and binding lists. All of the actual **var** parameters in a call must be *nonoverlapping*. If the actual parameters are simple names ("entire variables"), this requirement merely means that they must all be distinct. However, we must also prohibit passing a structured variable and one of its components (e.g., A and $A(1)$). What about two components of the same variable? This is allowed if they are distinct (e.g., $A(1)$ and $A(2)$), and disallowed if they are the same (e.g., $A(1)$ and $A(1)$). Since subscripts may be expressions, it may be necessary to generate a legality assertion (e.g., I **not=** J in the case of $A(I)$ and $A(J)$) to guarantee their distinctness.

It may appear that arrays already violate our rule that assignment to one entire variable can never change another. After all, assignment to $A(I)$ may change $A(J)$. However, these are not entire variables. We adopt the view of [Hoare and Wirth 1973, p.345] that an "assignment to an array component" is actually an assignment to the containing array. Thus $A(1) := 1$ is an assignment to A, and can be expected to change $A(J)$ if $J = 1$.

Pointers appeared to pose a more difficult problem. Assignment to $p\uparrow$ (i.e., to the variable to which p refers) may change the value of $q\uparrow$ (if p and q happen to point to the same variable, i.e., if $p = q$), or may even change the value of x (if pointer variables are allowed to point at program variables). We avoided the latter problem by retaining Pascal's restriction that pointers may only point to dynamically generated (anonymous) variables. (This is enforced by not providing an "address of" operator or coercion.) The usual treatment of the former problem is to consider pointers as indices into "implicit arrays" (one for each type of dynamic variable), and dereferencing as subscripting [Luckham and Suzuki 1976, Wegbreit and Spitzen 1976]. Thus $p\uparrow$ is merely a shorthand for $C(p)$, where C denotes p's implicit array, and the proof rules for arrays can be carried over directly. In particular, assignment via a dereferenced pointer is considered to be an assignment to its implicit array. From the verifier's standpoint, the situation is slightly better than that for arrays, since the decision of whether two subscripts are equal may involve arbitrary arithmetic expressions, while the decision of whether two pointers are equal reduces to the question of whether they resulted from the same dynamic variable generation ("New" invocation).

We have not yet discussed dereferenced pointers as variable parameters. If $p\uparrow$ and $q\uparrow$ (really $C(p)$ and $C(q)$) are both passed, the nonoverlapping requirement demands p **not=** q. Passing both p and $p\uparrow$ (really p and $C(p)$) is not a problem unless the formal parameter corresponding to p is dereferenced, which can only happen if C is accessible (i.e., imported). But then there would be an overlap between $C(p)$ and C, which makes the call clearly illegal. Passing pointers themselves as parameters (like passing array indices) does not create aliasing problems, since dereferenced pointers (like subscripted arrays) are not entire variables; assignment to one of them is considered as assignment to its implicit array.

Although the solution in the previous paragraph is formally complete, it is unsatisfactory in practice. The minor difficulty is that Euclid provides no way of naming implicit arrays for purposes of importation. The major problem is that it is too restrictive. It prohibits passing a dereferenced pointer as a variable parameter to any procedure that is allowed to dereference pointers to variables of the same type (i.e., that imports the implicit array for that type). We have solved both these problems by introducing the notion of *collections*, which are explicit program variables that act like the "implicit arrays" indexed by pointers. Each pointer is limited to a single collection, and $p\uparrow$ is still an acceptable shorthand for $C(p)$, where C is now the collection name. $p\uparrow$ is only allowed where C is accessible. Note that this makes it possible to pass pointers as parameters to procedures that are not allowed to dereference them, although they can copy them.

We allow any number of collections to have elements of the same type, with no more difficulty than arises from multiple arrays of the same type. Thus, the programmer can partition his dynamic variables and pointers into separate collections to indicate some of his knowledge about how they will be used; the verifier is assured that pointers in different collections can never point to overlapping variables. The astute reader will have noted that we have returned to the "class variables" that were in the original Pascal, but dropped in the revised version.

Collections also provide convenient units for storage management. We have chosen to associate with each

collection both the decision of whether to reference-count, and the selection of the (system- or user-supplied) storage management module (called a *zone*) to provide the space.

One consequence of our complete elimination of aliasing is that "value-result" and "reference" are completely equivalent implementation mechanisms for variable parameters, and a compiler is free to choose between them strictly on the basis of efficiency.

Modules

Since the introduction of "classes" by Simula 67 [Dahl et al. 1968], several programming languages have introduced mechanisms for "data encapsulation" or "information hiding" [Parnas 1971]. A survey of desirable properties of such mechanisms is given in [Horning 1976]. For Euclid, we chose something less powerful than "classes," "forms" [Shaw et al. 1977], or "clusters" [Liskov et al. 1977]. Our *modules* are closely akin to, but somewhat more complex than, the "modules" of Modula [Wirth 1977]. Adjusting the details of modules satisfactorily has been more difficult than expected. Perhaps this is because we still have an imperfect understanding, but it may also be because we violated our usual practice, and started from implementation considerations, rather than from verification issues.

The basic idea is that a module should "package up" a data structure and a related set of routines for its manipulation, and should hide its internal details from the outside world. We originally viewed it as a sort of glorified record, with some extra components (routines, types, initialization, finalization) and some control over the external visibility of its names (an export list). Like *record*, *module* is a type constructor, and a module type can be used to create many instances; this is the major source of differences between Euclid and Modula "modules."

Modules provide natural units for program construction. In fact, Euclid programs take the form of modules, rather than procedures; this is particularly appropriate when the program is to provide a number of entry points sharing a common data base that is to survive the various invocations (e.g., an operating system kernel). The "protection" provided by control over exported names serves as a useful first step towards abstract data types [Sigplan 1976]. In addition, they are used within the language in two places where it seemed important to effect a separation of concerns. The first is in iteration, where the knowledge of *how* to enumerate the elements of some data type should generally be associated with the type (module), rather than with each loop that needs such an enumeration. The problem, and its solution using *generators* is discussed in more detail in [Shaw et al. 1977]. We have chosen to use a simplification of the Alphard solution that seems powerful enough for the most common cases.

Similarly, the issues of *how* to allocate storage are quite separable from the uses to which that storage is put. We have chosen to isolate that knowledge in *zones*, which are (system- or user-defined) modules solely concerned with allocating and deallocating storage and ensuring that storage allocations never overlap. A zone deals with blocks of "raw storage"; it is the compiler's responsibility to ensure that its procedures are invoked at proper times, with correct parameters, and that the storage it allocates is properly initialized for its intended use, and that there is no type confusion or variable overlap outside the zone.

Types

One of the principal contributions of Pascal was its development of the notion of data types. Despite certain deficiencies [Habermann 1973], we find it more satisfactory than competitive approaches (e.g., the *modes* of Algol 68 [van Wijngaarden et al. 1976]). Pascal's types provide a flexible and convenient set of efficient data structuring mechanisms, and are useful conceptual tools for partitioning and organizing data within programs. In a type-rich language, such as Pascal, type-checking serves as a very effective compile-time error screen [Gannon and Horning 1975] [Gannon 1977].

It is a major undertaking to develop a new approach to data types that is both consistent and useful, and we did not attempt to do so within Euclid. Nevertheless, we felt compelled to try some small changes in the directions of safety and flexibility. Even these were difficult to get right.

Almost all type-checking in Pascal can be done at compile-time; the major exceptions are due to variant records and to the incomplete specification of formal parameters that are functions and procedures [Fischer and LeBlanc 1977]. The former are not a problem in Euclid, since such parameters are disallowed, but Euclid retains variant records. The problems in Pascal arise from aliasing (which we have already dealt with), from the treatment of the *tag* (which selects the current variant) as an ordinary, assignable field of a variant record, and from the accessibility of variant field selectors even when they do not apply to the current variant.

In Pascal, uncontrolled assignment to the tag field can change the current variant without ensuring that the corresponding fields contain values of appropriate types. We have eliminated this possibility in Euclid by making the tag a constant component of a variant record, and hence not assignable. If a variable is of variant record type, its current variant can only be changed by assignment of a record of one of the other variant record types; this assignment supplies a complete set of fields appropriate to that variant.

Variant field selectors are only accessible within the alternatives of a *discriminating case* statement, where the alternative is selected by the current tag. In the case statement, a local name is provided for the variant record (either as a constant or a variable); within any alternative, that name has the (nonvariant) type selected by the corresponding tag value, and all field selectors of that type are accessible. If the local name is bound to a variant record variable, the nonaliasing rule makes its more global name unusable in the scope; hence, there is no danger that its type may be changed within the scope (e.g., by calling a procedure that does so surreptitiously). If the local name is a constant, the variable may still be changed, but this will not affect the (discriminated) constant in any way, so access to its fields remains safe. Thus, variant records cannot be used to circumvent Euclid's type-checking. As a minor benefit, we avoid the need for the Pascal restriction that the same field names may not be used in separate variants.

Pascal treats (sub)ranges as types, and requires that all bounds be known at compile-time (i.e., be manifest constants). This is somewhat irksome for array bounds, and almost intolerable for routines that take array parameters. However, it allows a number of simplifications throughout the language, compiler, and verification system. We have allowed only a minor relaxation: bounds must still be constants, but they need not be manifest. In particular, a constant formal parameter of a routine may be used to specify a bound of another formal parameter. This will require verification that the bounds for the latter parameter are correct in all calls to the procedure since they are not fixed at compile time. We expect this usage to be common, and have supplied a shorthand; if a bound is specified as *parameter*, an additional (implicit) actual parameter containing the actual bound will be supplied automatically for each call.

A type declaration in Pascal provides a shorthand for a single type. In Euclid, a type declaration may have formal parameters. A parameterized declaration represents a set of

types, one of which is specified (by supplying actual parameters for all the formals) each time the type is referenced (e.g., to declare a variable). This allows the relationships among similar types to be made explicit in the program, and makes it easier for the program to exploit such relationships. Variant record definitions will usually appear within parameterized type declarations, with the tag being one of the formal parameters. Each particular value supplied as the corresponding actual parameter in a reference to such a type will select a particular alternative, i.e., will yield a nonvariant record. This is a useful feature (not available in Pascal), but it is often desirable to defer the choice of a variant. This can be done by using the special actual parameter *any*, which specifies that the type contains all values of the types corresponding to any choice for the tag, i.e., that the variant may be changed dynamically, by assignment.

Collections of variant records allow another degree of freedom. It is possible to select a variant at the time a dynamic variable is allocated, and to disallow any changes of variant by assignment. This is done by using the special actual parameter *unknown* in defining the object type of a collection. For each such *unknown* parameter, every call of New must supply an additional actual parameter that specifies the variant of the new dynamic variable. Both *any* and *unknown* specifications will lead to the use of discriminating case statements for access to the variant parts of records.

The *Pascal Report* is not very explicit about when two types are "the same," and it is not always clear what type-checking is allowed (required). E.g., 1..10 and 2..11 define subrange types that (in some sense) are clearly different. But what is the type of 2, which could be assigned to a variable of either type? Are we to assume that there are some subtle "coercions" going on (as is hinted in [Hoare and Wirth 1973])? Another problem: If **type** *Miles* = 1..10, and **type** *Hours* = 1..10, are *Miles* and *Hours* "the same" type or not? If the answer is "yes," the programmer has not gotten any protection by using different type names for conceptually different types; if it is "no," how do we justify using the same addition operator for both, and how can we write a routine that would accept either as a parameter? Should we go to the Algol 68 extreme of treating as "the same" all types that have the same representation, completely ignoring programmer-supplied type names? (See [Habermann 1973] for further examples of the difficulty of reasoning strictly from the hints given in [Wirth 1971] and [Hoare and Wirth 1973].)

We decided that the rules for type-checking must be quite explicit in Euclid (i.e., we would rather be wrong than vague in our answers to these questions). We have devoted considerable effort to spelling them out clearly. Firstly, we separately specified two kinds of checking: in a binding (e.g., formal/actual correspondence for a variable parameter) the two types must be *the same* (defined below); in other contexts (e.g., assignment, constant definition, constant parameter, operands of operators) a value of one type must be *compatible* with another type (e.g., within the proper subrange). Secondly, we never associate a subrange type with a value, rather the value gets the containing type (e.g., integer). Thirdly (after toying with having *synonym* and *nonsynonym* type declarations), we decided not to treat type declarations as creating new types; a type name is *the same* as its definition. Fourthly, every module definition creates an *opaque* type (i.e., one whose internal structure is not visible); types exported from modules are also opaque. Opaque types are only *the same* if they are defined by the same piece of text (i.e., even identical definitions define distinct types); exported types are *the same* only if exported (with the same name) from the same instance of the module type. Finally, two references to a parameterized type are *the same* only if their actual parameters are equal (this may cause the generation of legality assertions).

Containment Of Machine Dependencies

Euclid contains most of the "escape hatches" (providing direct access to machine features) typical of system implementation languages [Mohll 1975]. There is provision for machine-code routine bodies, for placing variables at fixed addresses, for specifying the internal representation of a record, and for explicitly overriding type-checking. Many of these features are difficult to define formally, and all of them pose problems for verification. We have not solved most of these problems; we have merely provided a mechanism for containing their effects.

Some modules may be explicitly declared to be machine-dependent; these are the only modules that are allowed to contain the various machine-dependencies mentioned above, or to contain machine-dependent modules. Machine-dependent modules serve to textually isolate these features, and to encapsulate their use; they may be imported into modules that are not machine-dependent (and rely only on the specifications, not the implementations of the imported modules). This does not simplify the process of verifying that machine-dependent modules actually do meet their specifications; it merely means that the verification of all other modules can proceed in a machine-independent manner.

We expect machine-dependent modules to be used for two different purposes: to provide efficient machine-dependent implementations for packages whose specification is machine-independent (e.g., string manipulation, high-level I/O), and to provide controlled access to machine features (e.g., channels, clocks, page tables). Programs using only the former should be quite portable, requiring changes to (and reverification of) only the bodies of the machine-dependent modules. However, in the latter case, machine-dependencies in the module specifications themselves will work against portability (which is not required for many of Euclid's intended applications, such as operating system kernels).

Conclusions

Even though Euclid does not represent a dramatic advance in the state of the art, we have accomplished several things relevant to reliability. Firstly, we have designed a useful language (Euclid minus machine-dependent modules), all of whose features are (in principle) verifiable in their full generality by existing techniques. Secondly, we have demonstrated that it is possible to completely eliminate aliasing in a practical programming language. Thirdly, we have made variant records completely type-safe.

By and large, the changes that we made to Pascal could be justified without reference to verification, and would be useful even in situations where verification is not a formal requirement. However, it is unlikely that many of them would have been made had verification not been one of our primary concerns. Furthermore, we seem to have been somewhat more successful at "getting it right the first time" when we started from a verification issue (e.g., nonaliasing, collections) than when we "worked back" from the implementation (e.g., modules, zones). Perhaps this is because the construction of proof rules is a useful discipline that makes it necessary to be very explicit about the interactions of language features.

This paper has not been able to convey the extent to which various design decisions were interdependent. None of them was made in isolation, and some of them caused ripples throughout the language. We feel good about the decision to make the control of visibility explicit, for example, because it supported so many of the other changes we made. The decisions to totally ban side effects in functions was triggered by an observation concerning legality assertions. It was the introduction of generators that reconciled some of us to the elimination of functions and procedures as parameters.

We are all reasonably happy with the way that Euclid has turned out. However, it is appropriate to ask how well it meets our original goals. Among other things, we were asked to "make *minimal* changes and extensions to Pascal," and our effort was to be "quite limited: only a month or two in duration." Even though we did not satisfy either goal, in retrospect it seems that both were quite necessary for whatever success we have had. It has taken us a year to carefully work through and document the interactions of the small set of changes to Pascal that we agreed to in the first two days; had we been more ambitious at the start, we would still be discovering surprising implications of "innocuous" changes.

It is hard to feel guilty about making more than minimal changes to the form of Pascal. As we have stressed in this paper, the conceptual changes have been relatively small; however, we expect them to lead to significantly different programming styles. Euclid is a language with its own "flavor" and style. It would be as wrong to try to cast it as "pidgin Pascal" as it would have been to cast Pascal as "pidgin Algol."

Finally, a few comments on language design by committee: It is not easy, under the best of circumstances. It is clear that any one of us could have designed a new language by himself with less effort than he expended on Euclid; it is equally clear that each of those languages would have contained hidden problems or limitations that we managed to expose and eliminate in the process of designing Euclid. The substantial variety in our backgrounds was very helpful in the design process, although it could have been a major stumbling block had we not started with a highly compatible set of views on what needed to be done. Surprisingly, our geographical distribution, which could have been expected to be an obstacle to close cooperation, was turned into an asset by the Arpanet. It made rapid communication convenient, and encouraged both five-way interaction on all issues and the maintenance of a complete record of all "discussions."

We surprised ourselves by spending much more time on "exposition" (writing the defining report and proof rules) than we spent on "language design." The latter would have been useless without the former, and it could be argued that the design will not be complete until we are satisfied with the exposition, but we somehow hadn't planned to spend so much time explaining. Conceptual unity in a report cannot be obtained by having everyone write a few sections; we found no substitute for having a single person (Butler Lampson) write and edit the entire defining document, with the advice and consent of the rest.

Acknowledgements

Obviously, we are greatly indebted to Wirth, whose Pascal language formed the principal basis of our work. We have also relied heavily on Hoare's work in the areas of programming language design, axiomatic methods, and program verification. We have consciously borrowed ideas from most of the languages represented at this conference, and have probably been influenced by many other languages and suggestions for language features. We have benefitted from comments and criticisms on the various drafts of the *Euclid Report* that have been provided by colleagues too numerous to mention here. Lauer and the other implementors have been particularly helpful in pointing out inadequacies of our design and exposition. Guttag, as an author of the proof rules, has similarly helped us. Our work has been significantly aided by the Arpanet, which allowed us to maintain effective and rapid communication in stating and resolving problems (and in maintaining a permanent record of such "discussions"), despite the wide geographical distribution of the authors. Lastly, both the *Euclid Report* and this paper owe much to Gail Pilkington's expert use of a computer editing and formatting system; the visual quality of both documents compelled us to work hard on their contents so that the beauty would be more than ink deep.

References

[Ambler et al. 1977] A. L. Ambler et al., "Gypsy: A Language for Specification and Implementation of Verifiable Programs," in [LDRS 1977].

[Ashcroft 1976] E. A. Ashcroft, M. Clint, and C. A. R. Hoare, "Remarks on 'Program Proving: Jumps and Functions' by M. Clint and C. A. R. Hoare," *Acta Informatica* **6**, pp. 317-318.

[Barnard and Elliott 1977] D. Barnard and D. Elliott (eds.), "Notes on Euclid," University of Toronto, Computer Systems Research Group Technical Report.

[Dahl et al. 1968] O.-J. Dahl et al., *The Simula 67 Common Base Language*, Norwegian Computer Center, Oslo.

[Dijkstra 1976] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall.

[Fischer and LeBlanc 1977] C. N. Fischer and R. J. LeBlanc, "Efficient Implementation and Optimization of Run-Time Checking in Pascal," in [LDRS 1977].

[Gannon and Horning 1975] J. D. Gannon and J. J. Horning, "Language Design for Programming Reliability," *IEEE Transactions on Software Engineering* SE-**1**, 2, pp. 179-191.

[Gannon 1977] J. D. Gannon, "An Experimental Evaluation of the Effect of Data Types on Programming Reliability," in [LDRS 1977].

[Gerhart and Yelowitz 1976] S. L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Transactions on Software Engineering* SE-**2**, 3, pp. 195-207.

[Goodenough 1975] J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Communications of the ACM*, **18**, 12, pp. 683-696.

[Habermann 1973] A. N. Habermann, "Critical Comments on the Programming Language Pascal," *Acta Informatica* **3**, pp. 47-57.

[Hoare 1973] C. A. R. Hoare, "Hints on Programming Language Design," *ACM Symposium on the Principles of Programming Languages*, Boston, pp. 1-30. (Also published as Stanford Computer Science Technical Report STAN-CS-73-403.)

[Hoare and Wirth 1973] C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica* **2**, pp. 335-355.

[Hoare 1975] C. A. R. Hoare, "Data Reliability," 1975 International Conference on Reliable Software, Los Angeles, pp. 528-533. (*SIGPLAN Notices* **10**, 6)

[Horning 1976] J. J. Horning, "Some Desirable Properties of Data Abstraction Facilities," *SIGPLAN Notices* **11**, 2.

[Lampson et al. 1977] B. W. Lampson et al., "Report on the Programming Language Euclid," *SIGPLAN Notices* **12**, 2.

[Lauer 1977] Further information may be obtained from H. C. Lauer, System Development Corporation, 2500 Colorado Avenue, Santa Monica, California.

[LDRS 1977] "Proceedings of a Conference on Language Design for Reliable Software," *SIGPLAN Notices* **12**, 3.

[Liskov et al. 1977] B. Liskov, et al., "Abstraction Mechanisms in CLU," in [LDRS 1977].

[London et al. 1977] R. L. London et al., "Proof Rules for the Programming Language Euclid," in preparation.

[Luckham and Suzuki 1976] D. Luckham and N. Suzuki, "Automatic Program Verification V: Verification-Oriented Proof Rules for Arrays, Records and Pointers," Stanford AI Lab Memo AIM-278, Stanford Computer Science Technical Report STAN-CS-76-549.

[McLaren 1977] M. D. McLaren, "Exception Handling in PL/I," in [LDRS 1977].

[Melliar-Smith and Randell 1977] P. M. Melliar-Smith and B. Randell, "Software Reliability: The Role of Programmed Exception Handling," in [LDRS 1977].

[Mohll 1975] W. L. van der Poel and I. Maarssen (eds.), *Machine Oriented Higher Level Languages*, North-Holland/American Elsevier.

[Parnas 1971] D. L. Parnas, "Information Distribution Aspects of Design Methodology," *Proceedings of IFIP Congress 71*, North-Holland, pp. 339-344.

[Popek 1976] G. J. Popek, Arpanet message, 6 Jan. 1976.

[Shaw et al. 1977] M. Shaw et al., "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," in [LDRS 1977].

[Sigplan 1976] "Proceedings of Conference on Data: Abstraction, Definition, and Structure," *SIGPLAN Notices*, **11**, 2.

[Wegbreit and Spitzen 1976] B. Wegbreit and J. Spitzen, Proving Properties of Complex Data Structures, *Journal of the ACM*, **23**, 2 pp. 389-396.

[van Wijngaarden et al. 1976] A. van Wijngaarden (ed.) et al., *Revised Report on the Algorithmic Language ALGOL 68*, Springer-Verlag, Berlin, New York.

[Wirth 1971] N. Wirth, "The Programming Language Pascal," *Acta Informatica* **1**, pp. 35-63.

[Wirth 1977] N. Wirth, "Towards a Discipline of Real-Time Programming," in [LDRS 1977].

[Wulf et al. 1977] W. A. Wulf, M. Shaw, and R. L. London, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Transactions on Software Engineering* SE-2, 4, pp. 253-265.

[Wulf and Shaw 1973] W. A. Wulf and M. Shaw, "Global Variables Considered Harmful," *SIGPLAN Notices* **8**, 2, pp. 28-34.