A STUDY OF PROTECTION IN PROGRAMMING LANGUAGES

Allen L. Ambler
Amdahl Corporation
and
Charles G. Hoch
The University of Texas at Austin

The concept of "protection" in programming languages refers to the ability to express directly in the language the desired access control relationships for all objects defined in the language. The use of such mechanisms as data types, scope, parameter passing mechanisms, routines as parameters, abstract data types, and capabilities in Pascal, Concurrent Pascal, Euclid, Clu, and Gypsy are explored via a simple example which embodies many protection problems. The usefulness of language defined and enforced protection mechanisms to the process of formal verification is discussed.

Keywords and Phrases: Protection, Pascal, Concurrent Pascal, Euclid, Clu, Gypsy, abstract data types, module

CR Catagories: 4.22, 4.34

Introduction

The concept of "protection" in programming languages refers to the ability to express directly in the language the desired access control relationships for all objects defined in the language. While in operating systems protection has received extensive attention, only recently have we recognized the need for similar protection facilities in programming languages. There are several factors which have contributed to this recent realization. First, the emphasis being placed on producing better software has generally supported the concept of highly modular programs with well-defined and tightly-enforced module interfaces. Parnas modules [10], step-wise refinement [11], structured programming [3], etc. all employ modules with rigid interface conventions which whenever possible are enforced by the language and which when unenforceable rely on self-imposed programmer discipline. Second, formal program verification either by hand or by semi-automated techniques must rely on information about object accessibility in order to prove assertions about manipulations on objects. These object access restrictions necessary for formal verification need to be expressed directly in the language and to be enforceable by compilers of that language.

As various protection needs have arisen new protection mechanisms have been introduced to handle them. Data types have been introduced to protect objects from misinterpretation across module interfaces. Scope of variables allows selective "hiding" of objects; thereby, limiting access. Independent compilation provides another means of selectively "hiding" objects. Parameter passing mechanisms provide for "partial" access to variables passed across module interfaces. Routines as parameters provide a means of passing indirect access to an object without allowing direct, uncontrolled access. The implementations of abstract data types provide new mechanisms for selectively "hiding" objects and for allowing "partial" access. Finally, there have been proposals for introducing "capabilities" into programming languages [6].

In the course of this paper we will explore the usage of these protection mechanisms as presented in specific existing and/or future languages. The vehicle for our discussions will be an example, referred to as the Prison Mail System.

Prison Mail System Problem

The Prison Mail System. (PMS) Problem is described as follows:

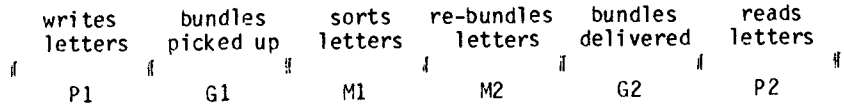

Figure 1. Bundle Cycle

1. Prisoners are confined to individual cells unable to communicate directly or indirectly with any of the other prisoners, except via the Prison Mail System.

2. Prisoners communicate through the Prison Mail System by exchanging messages which are picked-up and delivered by the prison guards.

3. Each message consists of a body and a signature and is placed inside an envelope which is then sealed and to which an address is affixed.

The Prison Mail System resembles the Post Office System except that the guards and the prisoners are mutually suspicious of each other. The guards are afraid that the prisoners may be planning a prison break and consequently, are constantly alert for any information that might reveal the prisoners' plans. The prisoners, on the other hand, resent this snooping by the guards and do everything in their power to make it impossible for the guards to obtain information. By mutual agreement there is one and only one security constraint which is always observed.

The Security Constraint: Only the addressee is allowed to open an envelope once sealed.

However, the guards are allowed to remember any information that is discernible by examination including from whom and to whom messages are carried. Cognizant of this fact, the prisoners worked out the following convention for reducing the significance of the information obtainable by the guards.

4. For each prisoner all messages (possibly none) to be mailed during some time period are placed inside another envelope, referred to as a bundle, which is then sealed and labelled for a non-partisan Postmaster.

5. The Postmaster receives bundles, opens them, sorts the letters contained inside (without opening them), rewraps them into a single bundle (possibly empty) for each prisoner, and relabels them for the correct recipient.

Hence, during each delivery cycle each prisoner sends to and receives from the Postmaster exactly one bundle. Thus the guards are thwarted from discovering which prisoners are communicating. We choose to ignore any further possibilities of discovery such as weighing the bundles, measuring their thickness, etc.

A PMS Solution

In the remainder of this paper we will discuss a solution to the PMS problem as it is expressible in the languages Pascal [5], Concurrent Pascal [2], Euclid [7], Clu [8], and Gypsy [1]. For each language we will analyze the effectiveness of its protection mechanisms at enforcing the Security Constraint. We note from the outset that none of these languages, except Gypsy, was specifically designed for protection. It is not our intention to criticize them in any way, but rather to contrast the various existing protection mechanisms. It is our contention that protection in programming languages is an old, but real, issue that has often been buried in other constructs and that only recently has been addressed directly. We begin by characterizing the general properties of the solution.

If we limit our thoughts to a single "bundle" cycle, we observe that (P1) the prisoners write letters wrapping them in a bundle, (G1) the guards pick up the bundles and deliver them to the Postmaster, (M1) the Postmaster opens the bundles and sorts the mail, (M2) the Postmaster then re-bundles the letters and hands them back to the guards, (G2) the guards deliver the bundles to the prisoners, and (P2) the prisoners open the bundles and read their mail. The "bundle" cycle is illustrated in Figure 1. We now write individual algorithms for each phase of the "bundle" cycle.

Algorithm P1: Start with an empty bundle and while the mood strikes compose letters signing each before placing it into an envelope and addressing it. When the mood no longer persists, seal the bundle, affix the Postmaster's name to the label, and place the bundle in the mailbox.

Algorithm G1: From each prisoner pick up one bundle from his mailbox addressed to the Postmaster and deliver them to the Postmaster's mailbox.

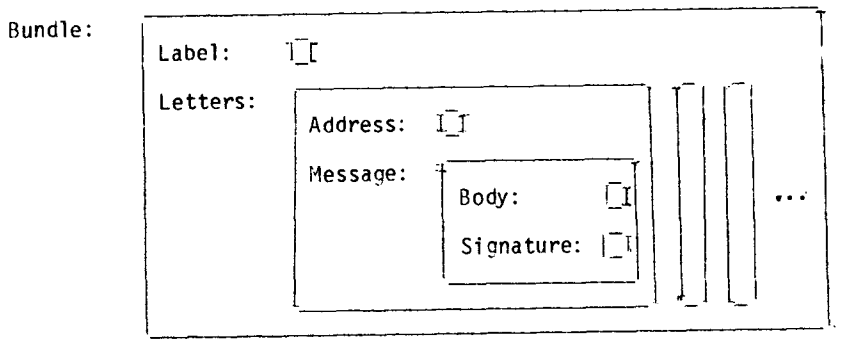

Figure 2. PMS Bundle/Letter/Message Structure

Algorithm M1: While there are bundles in the Postmaster's mailbox remove one and sort the letters into piles - one pile for each prisoner.

Algorithm M2: For each prisoner, place his pile of letters into an empty bundle, seal the bundle, affix the prisoner's name to the label, and place it in the Postmaster's mailbox.

Algorithm G2: Pick up the bundles from the Postmaster's mailbox and deliver them to the labelled prisoner's mailbox.

Algorithm P2: Remove the bundle from the mailbox, open it, and while the bundle is not empty remove, open, and read the letters.

We assume a data structure as illustrated in Figure 2 where each "Bundle" is composed of a "Label" and a sequence of "Letters" - where each of the latter is composed of a "Address" and a "Message" - and where the latter is composed of a "Body" and a "Signature". Thus each bundle is composed of four elementary fields. It is the "selective" protection of these four elementary fields that constitutes the heart of this example problem.

By analyzing the six algorithms in terms of their required access to each of these four fields we arrive at the access matrix presented as Figure 3. Two categories of protection are indicated: (N) access is necessary for the algorithm to function and (P) access should be prevented as the algorithm does not need access. For each of the "necessary" accesses a "/R" or "/W" is included to indicate that either "read" or "write" access is required. In the Figure 3 there are several instances where "P or N/R" access is indicated. For these entries either choice is allowed because, while the algorithm does not need access, the information is already available and a suitable choice will for certain languages simplify the solution.

What remains to be discussed for our solution is the flow of control within the program. Figure 4 models the interrelationships between the prisoners, the guards, and the Postmaster. The internal communication structure for the guards is unimportant for our purposes and is left unspecified.

A simple control solution might be:

```
program
    var Boxes
    procedure P1
    procedure G1
    procedure M1
    procedure M2
    procedure G2
    procedure P2
    loop
        P1
        G1
        M1
        M2
```

	Label	Address	Body	Signature
P1	N/W	N/W	N/W	N/W
G1	N/R	P	P	P
M1	P or N/R	N/R	P	P
M2	N/W	P or N/R	P	P
G2	N/R	P	P	P
P2	P or N/R	P or N/R	N/R	N/R

Figure 3. PMS Access Chart

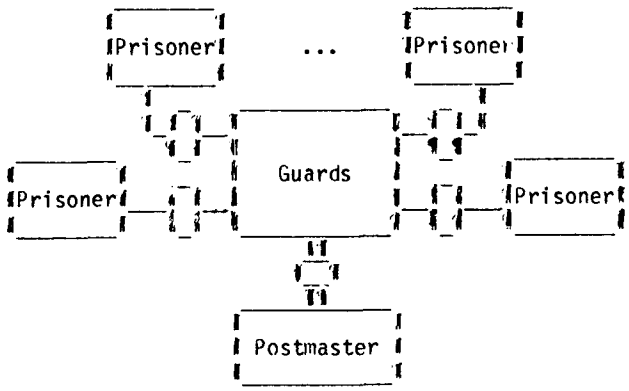

Figure 4. PMS Process Structure

```
        G2
        P2
    end
```

However, as we will see protection is often tied explicitly or implicitly to control flow and will dictate more complex control solutions. These will be discussed as they are encountered.

Liberties

In the solutions that follow we will take certain liberties to create shorter, more understandable solutions. We assert that these liberties merely eliminate detail from our discussion and in no way add to or detract from the languages' ability to express the essential properties of this protection problem.

- NPrisoners is an integer constant specifying the number of prisoners using the PMS.

- PrisonerId is a range or set of prisoner names which may be thought of as integer values in the range 1..NPrisoners.

- sequence of ... is an allowable type which will be treated as unbounded. It has operations append item to sequence, remove item from sequence, and empty(sequence). Append inserts a new item on the tail of the sequence and remove removes the head item from the sequence (FCFS). Empty is a boolean function that evaluates to true when the sequence is empty.

- String is a sequence of character.

- loop ... end is a control statement that indicates an infinite loop. All the solutions to the PMS problem presented in this paper will deliberately not terminate.

- for i:= PrisonerId do ... is a finite loop construct that causes the do clause to be executed exactly once for each value of type PrisonerId.

- decoy is a dummy type used solely to obscure a previous definition.

Note: For the purpose of the solutions that follow, it makes no difference whether we consider that the Postmaster is also a prisoner or not.

Solution in Pascal

We chose to begin by looking at Pascal [5] because it is representative of the protection facilities present in most algorithmic programming languages and because it has had a strong influence on many of the languages we will examine subsequently.

Pascal offers data types, scope rules, parameter passing limitations, and routines as parameters for enforcing access protection. Data type checking assures that the types of all actual parameters to all functions, procedures, and operators (including assignment) "match" the corresponding formal parameters. "Matching" is interpreted to mean that they have the same structure, not necessarily the same type name. This interpretation has strong implications for protection. It means that no matter how much protection is wrapped around a type it is always possible to "impersonate" the type by defining a different type of the same structure. Pascal then allows assignment of the "protected" object to the "impersonating" object. This allows the contents of the "protected" object to be discovered.

```
procedure A
    type t= yyy
    var a: t
    end;
procedure B
    var b: t
    end;
```

5a

```
type t= yyy

procedure A
   var a: t
  end;
procedure XB
   type t= decoy

    procedure B
        var b: t
        end;
    end;
```

5b

Figure 5. Scope Protection

Pascal has a typical hierarchical program structure which allows for nested routine declarations with the accompanying scope rules: 1. an identifier has scope over the defining routine and all encompassed routines and 2. where an identifier is multiply defined in a scope then references to the identifier are resolved by the definition with the smallest encompassing scope. Rule 1 means that objects defined in a particular scope are protected from references from unencompassed scopes. Rule 2 means that objects defined in encompassing scopes can be "hidden" if they are redefined in smaller, but still encompassing, scopes. The two scope rules are illustrated in Figure 5. In 5a type "t" is hidden from procedure "B" by keeping the definition out of the scope of "B"; while in 5b, type "t" is hidden from "B" by inserting an intervening "decoy" definition in procedure "XB".

Pascal allows call by value as a parameter passing restriction to prevent undesired actual parameter modification. However, this mechanism is unable to prevent unwarranted actual parameter examination. In addition, any parameter which is also a global variable can be modified directly without restriction.

Finally, Pascal allows routines to be passed as parameters to other routines. Unfortunately in so doing, the need is created for dynamic parameter type checking. Pascal implementations typically don't do this dynamic checking, but we will ignore this bit of laziness. The capability to pass routines as parameters, combined with the scope rules, allows an object X to be protected from a routine R by defining X out of the scope of R (hence, preventing direct access to X) and by passing R only routines which can manipulate X - never X itself. However, as we shall see in the PMS solution this technique can get awkward.

The complete solution expressed in Pascal is presented at the end of this section. This solution has the following form.

```
program
    procedure M1M2
    procedure G1
    procedure G2
    procedure P1
    procedure P2
    procedure XMain
        var Boxes
        loop
            P1
            G1
            M1M2
            G2
            P2
        end
    XMain
```

To understand why the solution takes this form consider the following arguments:

A. Suppose G1/G2 have direct access to Boxes either because Boxes is global to G1/G2 or because Boxes is passed as a parameter to G1/G2. Then :

   a. If G1/G2 also have access to all of the type definitions for Boxes, then G1/G2 can access the contents of Boxes and the Security Constraint is violated.

   b. If G1/G2 have access to only some or even none of the type definitions, they can still fabricate structurally equivalent declarations for the missing definitions, then declare a local variable using the fabricated types and assign to that local variable the contents of Boxes; thereby, being able to violate the Security Constraint.

   Hence Pascal's treatment of types implies that if G1/G2 have direct access to Boxes then the Security

```
program Main;
type MessageType =
    record Body: String; Signature: PrisonerId; end;
type LetterType =
    record Address: PrisonerId; Message: MessageType; end;
type BundleType =
    record Label: PrisonerId; Letters: sequence of LetterType; end;
type Mainbox = sequence of BundleType;
type Mailboxes = array [PrisonerId] of BundleType;
procedure M1M2(function EmptyMasterBox, EmptyScratchBundle: boolean; function
        ReadAddressScratchLetter: PrisonerId; procedure RemoveMasterBox, AppendMasterBox,
        RemoveScratchBundle, AppendScratchBundles, MoveScratchBundles, WriteLabelScratchBundle);
        ...
    end M1M2;
procedure G1(procedure RemoveBoxes, AppendMasterBox);
        ...
    end G1;
procedure G2(function EmptyMasterBox: boolean; function
        ReadLabelScratchBundle: PrisonerId; procedure RemoveMasterBox, AppendBoxes);
        ...
    end G2;
procedure P1(var b: BundleType);
        ...
    end P1;
procedure P2(var b: BundleType);
        ...
    end P2;
procedure XMain;
    var Boxes: Mailboxes;
    var MasterBox: Mainbox;
    var ScratchBundle: BundleType;
    var ScratchBundles: array [PrisonerId] of BundleType;
    var ScratchLetter: LetterType;
    function EmptyMasterBox: boolean;
        begin EmptyMasterBox:= empty(MasterBox); end;
    procedure RemoveMasterBox;
        begin remove ScratchBundle from MasterBox; end;
    procedure AppendMasterBox;
        begin append ScratchBundle to MasterBox; end;
    procedure RemoveBoxes(i: PrisonerId);
        begin ScratchBundle:= Boxes[i]; end;
    procedure AppendBoxes(i: PrisonerId);
        begin Boxes[i]:= ScratchBundle; end;
    function EmptyScratchBundle: boolean;
        begin EmptyScratchBundle:= empty(ScratchBundle); end;
    procedure RemoveScratchBundle;
        begin remove ScratchLetter from ScratchBundle; end;
    procedure AppendScratchBundles(i: PrisonerId);
        begin append ScratchLetter to ScratchBundles[i]; end;
    procedure MoveScratchBundles(i: PrisonerId);
        begin ScratchBundle:= ScratchBundles[i]; end;
    function ReadLabelScratchBundle: PrisonerId;
        begin ReadLabelScratchBundle:= ScratchBundle.Label; end;
    procedure WriteLabelScratchBundle(i: PrisonerId);
        begin ScratchBundle.Label:= i; end;
    function ReadAddressScratchLetter: PrisonerId;
        begin ReadAddressScratchLetter:= ScratchLetter.Address; end;
    begin
        loop
            for i:= PrisonerId do P1(Boxes[i]);
            G1(RemoveBoxes, AppendMasterBox);
            M1M2(EmptyMasterBox, EmptyScratchBundle, ReadAddressScratchLetter, RemoveMasterBox,
                AppendMasterBox, RemoveScratchBundle, AppendScratchBundles,
                MoveScratchBundles, WriteLabelScratchBundle);
            G2(EmptyMasterBox, ReadLabelScratchBundle, RemoveMasterBox, AppendBoxes);
            for i:= PrisonerId do P2(Boxes[i]);
        end;
    end;
begin XMain; end.
```

Figure 6. Pascal Solution

Constraint can be violated.

B. Suppose G1/G2 are not allowed direct access to Boxes, then it must be: 1. that Boxes is defined in such a way that its scope does not include G1/G2, 2. that Boxes is not passed as a parameter to G1/G2, and 3. that G1/G2 are instead passed routines, as parameters, which can access Boxes. Then:

   a. If Boxes is defined outside the scope of G1/G2 as in Figure 5a the result takes the form presented here.

   b. Otherwise, if Boxes is defined so that its scope encompasses G1/G2 then there must be an intervening "decoy" definition as in Figure 5b. This possibility likewise leads to a valid solution. The solution presented for Concurrent Pascal uses this technique.

In the solution, the Main program simply functions as a shell and immediately calls XMain which contains the data declarations safely out of reach. Since the data within a bundle need not be protected from the prisoners, P1/P2 are allowed direct access to their parameters. To prevent ever passing a letter (or structure of letters) to G1/M1M2/G2 and yet allow them to perform their functions, we were forced to create twelve access routines. Many of these functions perform the same operation, but on different objects (e.g. AppendBoxes and MoveScratchBundles) yet without the objects being passed as parameters they are all necessary. Variables that would normally be declared locally (such as ScratchBundles) within G1/M1M2/G2 are forced to be declared in XMain with their access protected in the same manner by using access functions. The net result is that G1/M1M2/G2 are only synthetic representations of their intended design.

Solution in Concurrent Pascal

The next protection mechanism we want to look at is the abstract data type. In its simplest form an abstract data type consists of a set of abstract operations defined on a concrete representation which is hidden from all references except those of the abstract operations. Simula 67 [4] provides a class structure which was initially defined as described above, except that the concrete representation was completely visible. A later extension [9] allows selective hiding of both data and routine declarations. The class mechanism defined in Concurrent Pascal [2] behaves similarly. In it all names are hidden, except those explicitly declared as entries. We have chosen to display the PMS solution in only one of these languages as the solution in the other is similar. Since the rest of the examples in this paper use a Pascal-like syntax we have chosen Concurrent Pascal.

Concurrent Pascal offers the same protection mechanisms available in Pascal plus classes and monitors. In the current context we will ignore the possibilities for a concurrent solution, and hence monitors. As stated above classes consist of a set of operations defined on a concrete representation of an object for the purpose of abstracting its essential properties. An important fact is that each class is unique, i.e. two classes "match" if and only if they are the same declaration. The class structure is illustrated in Figure 7. It defines an abstract IntStack with operations Push, Pop, and Top (ignoring all error possibilities). The keyword entry indicates that the identifier is to be defined in the scope of the class declaration. The final begin-end pair provides code which is to be executed whenever the class is allocated and which initializes the internal structure.

The solution expressed in Concurrent Pascal has the following form.

```
var Boxes
procedure XM
    procedure M1M2
    M1M2
procedure XG
    procedure G1
    procedure G2
    G1
    XM
    G2
procedure XP1
    procedure P1
    P1
procedure XP2
    procedure P2
    P2
loop
    XP1
    XG
    XP2
end
```

The solution functions by placing each of P1/G1/M1M2/G2/P2 in an environment where access functions are selectively screened by defining decoy definitions in intervening scopes. Type MessageType is screened from G1/M1M2/G2 by XM and XG, but not from P1/P2 and type LetterType is screened from G1/G2, but not P1/M1M2/P2.

In addition, MessageType, LetterType, and BundleType are defined as classes thereby restricting access to only the appropriate access functions. Note that a bit of cleverness has been employed here. For LetterType we want that M1M2 can read, but not write, the Address field only. By grouping the operations on letters such that one set can be performed without a MessageType parameter while the others cannot, access to the operations is divided into two categories: complete

```
type IntStack = class;
    var st: array [1:100] of integer;
    var pt: integer;
    procedure entry Push(i: integer);
        begin pt:= pt+1; st[pt]:= i; end;
    procedure entry Pop;
        begin pt:= pt-1; end
    function entry Top: integer;
        begin Top:= st[pt]; end;
    begin pt:= 0; end;
```

Figure 7. Concurrent Pascal Class

and partial. Those routines which cannot reference MessageType have only partial access and with partial access only ReadAddress is available. In this case M1M2 has partial access and P1/P2 have complete access. The same cleverness is employed in the definition of BundleType to allow G1/G2 partial access while allowing P1/M1M2/P2 complete access.

Solution in Euclid

Euclid [7], also derived from Pascal, was designed specifically for the expression of systems programs that are to be verified. Consequently, Euclid has specifically included features to control object accessibility. Many of the problems with which we were presented in Pascal, have been brought under programmer control.

The treatment of data types in Euclid is essentially identical to that of Concurrent Pascal with classes being replaced by modules. Two types "match" if they have the same structure, except that all module types are considered different. Hence, two types which contain modules can "match" only if they contain the identical module types is corresponding positions of otherwise "matching" structures. Thus by using module type definitions it is now possible to prevent the "impersonation" of types as we found in Pascal.

A transformation has been performed on the Pascal scope rules. While declarations are still hierarchically definable, the implied scope of a declaration is only the routine in which it is defined and not the encompassed routines as well. This means that without additional mechanisms there would be no non-local definitions (including routines). However, there are two mechanisms for extending the scope of declarations. The first is a pervasive declaration. If an object is declared pervasive, then its scope also includes all routines encompassed by the defining routine. This is equivalent to the Pascal scope, except that in the extended scope the object may not be modified. For types, routines, and constants there is no means of modifying them anyway, so declaring them pervasive restores the Pascal scope rules. But for variables, while their scope is extended, they may only be referenced as constants in the extended scope. The second mechanism is an imports clause which may be attached to type and routine declarations. The effect of an import statement is to extend the scope of those identifiers named in the imports clause to include the type or routine being defined. Names may only be imported one scope level; hence, if they are not imported at each level intervening between their definitions and a particular type or routine, then they become invisible to that type or routine. Variables may be imported either with full modification capabilities (as var) or as constants (the default). Figure 9 shows the effect of the pervasive and import rules. The variable "y" is available to each of the routines "A", "B", "C", and "D", but only as a constant. The variable "x" is available as a variable to "A", as a constant to "B", and is unavailable to both "C" and "D" (it is completely hidden from "D" in that it could not be imported even if desired).

Euclid's module is a variation of the abstract data type we found in Simula and Concurrent Pascal. The variables declared within the body are considered to be the concrete representation of the abstract object with the routines declared therein being the abstract operations. Those names which are to represent the abstract properties are then exported into the scope of the type declaration. Identifiers which are to be exported are explicitly listed in an export clause. Euclid furthers allows that variables may be exported as either variables (var) or constants (again the default). Note: Importing a type identifier imports all identifiers exported by that type definition.

Euclid does not permit procedures and functions to be passed as parameters, thus precluding the technique used in the Pascal solution.

The solution expressed in Euclid has the following form.

```
module main
    module bundletype
        module lettertype
            P1
            P2
        M1M2
        bundle_P1
        bundle_P2
```

```
type MessageType = class;
    var Body: String;
    var Signature: PrisonerId;
    function entry ReadBody: String;
        begin ReadBody:= Body; end;
    procedure entry WriteBody(s: String);
        begin Body:= s; end;
    function entry ReadSignature: PrisonerId;
        begin ReadSignature:= Signature; end;
    procedure entry WriteSignature(a: PrisonerId);
        begin Signature:= a; end;
    begin end;
type  LetterType = class;
    var Address: PrisonerId;
    var Message: MessageType;
    function entry ReadAddress: PrisonerId;
        begin ReadAddress:= Address; end;
    procedure entry ReadMessage(var m: MessageType);
        begin m:= Message; end;
    procedure entry WriteLetter(m: MessageType; a: PrisonerId);
        begin Message:= m; Address:= a; end;
    begin end;
type BundleType = class;
    var Label: PrisonerId;
    var Letters: sequence of LetterType;
    function entry ReadLabel: PrisonerId;
        begin ReadLabel:= Label; end;
    procedure entry WriteLabel(l: LetterType; a: PrisonerId);
        begin Label:= a; end;
    procedure entry EmptyLetters(l: LetterType);
        begin EmptyLetters:= empty(Letters) end;
    procedure entry RemoveLetter(var l: LetterType);
        begin remove l from Letters; end;
    procedure entry AppendLetter(l: LetterType);
        begin append l to Letters; end;
    begin end;
type Mainbox = sequence of BundleType;
type Mailboxes = array [PrisonerId] of BundleType;
var Boxes: Mailboxes;
procedure XM(var masterbox: Mainbox);
    type MessageType = decoy;
    var Boxes: decoy;
    procedure M1M2(var masterbox: Mainbox); ... end M1M2;
    begin M1M2(masterbox); end;
procedure XG(var boxes: Mailboxes);
    type MessageType = decoy;
    type LetterType = decoy;
    var MasterBox: Mainbox;
    procedure G1(var boxes: Mailboxes; var masterbox: Mainbox); ... end G1;
    procedure G2(var boxes: Mailboxes; var masterbox: Mainbox); ... end G2;
    begin init MasterBox; G1(boxes, MasterBox); XM(MasterBox); G2(boxes, MasterBox); end;
procedure XP1(var box: BundleType);
    var Boxes: decoy;
    procedure P1(var b: BundleType); ... end P1;
    begin P1(box); end;
procedure XP2(var box: BundleType);
    var Boxes: decoy;
    procedure P2(var b: BundleType); ... end P2;
    begin P2(box); end;
begin
    init Boxes;
    loop
        for i:= PrisonerId do XP1(Boxes[i]);
        XG(Boxes);
        for i:= PrisonerId do XP2(Boxes[i]);
    end;
end.
```

Figure 8. Concurrent Pascal Solution

```
var x: integer;
pervasive var y: integer;
procedure A imports (var x)
    procedure B imports (x)
        procedure C
            procedure D imports(?)
```

Figure 9. Pervasive and Import Rules

```
procedure G1
procedure main_M1M2
procedure G2
var boxes
var MasterBox
loop
    bundle_P1
    G1
    main_M1M2
    G2
    bundle_P2
end
```

This form is dictated by the need to hide certain identifiers from the guards and the postmaster. By declaring LetterType inside of BundleType, P1 and P2 inside the LetterType module, and the postmaster (M1M2) inside the BundleType module, the required data protection is established.

The data abstraction mechanism in Euclid allows a routine access to the internal representation of at most one instance of a given module type, via importation of module variables. Thus, a sequence of module instances must be broken down into the individual instances before an operation on the module can be done. P1 has access to the components of a letter by importing the variables which form an instance of a letter (that is, address and message). It generates a sequence of letters by recursive calls to itself. Since P1 can only be referenced as a component of an instance of a letter, bundle_P1 declares a variable of type LetterType. P1 would have the following form.

```
var letter: LetterType;
set_address;
set_message;
if moodstrikes
    then
        letter.P1;
        append letter to contents;
    end.
```

Similarly, M1M2 recursively processes bundles, being initially called by main_M1M2 with the head of MasterBox. Its form would be:

```
var bundle: BundleType
process_label;
process_contents;
if not empty (MasterBox)
    then
        remove bundle from MasterBox;
            bundle.M1M2;
    end;
process_label;
process_contents.
```

Finally P2 recursively consumes letters. It is initially called by bundle-P2 with the head of contents. Its form would be:

```
var letter: LetterType;
consume_address;
consume_message;
if not empty (contents)
    then
        remove letter from contents;
        letter.P2;
    end.
```

An important property of this solution is that in order to produce the desired data access control, the control structure of the program is not really correct. For example, nothing prevents guards (G1 and G2) from calling the prisoners and the postmaster, via calls to bundle_P1, bundle_P2, and M1M2, and the postmaster (M1M2) from calling the prisoners, via calls to P1 and P2. This is a consequence of the rule in Euclid that if a module name is known, all identifiers exported by the module are known.

Solution in Clu

The language Clu [8] displays significant departures from the preceding languages while retaining a Pascal-like syntax and scope rules. Clu was designed specifically to support the development of a program by successive decomposition through the usage of abstractions. An abstraction is for Clu "a mechanism which permits the expression of relevant details and the suppression of irrelevant details" [8].

The basic unit in Clu is a module. A module is either a procedure or a cluster. A procedure provides an abstract operation and a cluster provides an abstract object. Modules may be nested similar to Pascal, except that an identifier may not be redefined in an encompassed scope as in Pascal.

```
module Main
    pervasive type BundleType =
        module exports (:=, label, bundle_P1, M1M2, bundle_P2)
            pervasive type LetterType =
                module exports (:=, address, P1, P2)
                    type MessageType =
                        record
                            Body: String;
                            Signature: PrisonerId;
                        end;
                    var Address: PrisonerId;
                    var Message: MessageType;
                    procedure P1 (var label: PrisonerId, var contents: sequence of LetterType)
                            ....
                        end;
                    procedure P2(var label: PrisonerId, var contents: sequence of LetterType)
                            ....
                        end;
                    initially Message.body := ();
                end LetterType;
            var label: PrisonerId;
            var contents: sequence of LetterType;
            procedure M1M2 (var MasterBox: sequence of BundleType)
                    ....
                end;
            procedure bundle_P1 imports (var label, var contents) =
                begin
                var letter: LetterType;
                    letter.P1 (label, contents);
                    append letter to contents;
                end;
            procedure bundle_P2 imports (var label, var contents) =
                begin
                var letter: LetterType;
                    if not empty (contents)
                        then
                            remove letter from contents;
                            letter.P2 (label, contents);
                        end;
                end;
            initially contents := ();
        end BundleType;
    pervasive type MainBox = sequence of BundleType;
    pervasive type MailBoxes = array (PrisonerId) of BundleType;
    var MasterBox: MainBox;
    var Boxes: MailBoxes;
    procedure G1 (var Boxes: MailBoxes, var MasterBox: MainBox)
            ....
        end;
    procedure G2 (var boxes: MailBoxes, var MasterBox: MainBox)
            ....
        end;
    procedure main_M1M2 (var MasterBox: MainBox) =
        begin
        var bundle: BundleType;
            remove bundle from MasterBox;
            bundle.M1M2;
        end;
    loop
        for i: PrisonerId do Boxes[i].bundle_P1;
        G1 (Boxes, MasterBox);
        main_M1M2 (MasterBox);
        G2 (Boxes, MasterBox);
        for i: PrisonerId do Boxes[i].bundle_P2;
    end;
end Main;
```

Figure 10. Euclid Solution

A cluster is another form of an abstract data type. It contains an object representation which is used to realize the abstraction along with a set of operations defined on this representation, which provide the required abstract operations. The representation is known only within the cluster and therefore only referencible by the operations defined within the cluster. The set of operations to be exported into the scope of the cluster declaration is explicitly specified in the cluster heading.

A routine may return multiple values as the routine result and then perform a multiple assignment. For instance,

```
stack, element:= Pop(stack)
```

where

```
Pop = operation(s: stacktype)
        returns (stacktype, elementtype);
      ...
```

would allow the Pop operation to return the top value as well as the reduced stack.

Summarizing, Clu has Pascal-like scope rules with the restriction that unqualified identifiers cannot be redefined in an encompassed scope and it has an abstract data type, called a cluster, with explicit exportation. Hence, if we look back at the previous solutions, the solution expressed in Concurrent Pascal is unusable because redefining identifiers in encompassed scopes is not allowed. The solution in Euclid is usable, as is the solution expressed in Pascal. There is yet a more interesting solution which is unique to Clu.

A variable in Clu is an identifier capable of denoting objects of a certain specified type. A variable is made to denote a particular object by means of the assignment operation. An object is a structure capable of possessing a value of the specified type. Objects are dynamically created by the create operation defined for each type (either explicitly or implicitly). Each newly created object is unique, i.e. it has never before existed. Thus there are two distinct forms of "equality" in Clu. Two variables are equal if and only if they both denote the same object. Two variables are similar if they both denote objects which possess the same value. Because of the uniqueness of objects, if equal(X, Y) then there exists a Z which created the object now pointed to by both X and Y together with a sequence of assignments such that X:= ... := Z and Y:= ... := Z. The significance of this fact is that it is impossible to fabricate an object and then have it test equal to some other object. This gives us the ability to construct unique, unfabricatable protection keys.

The solution presented below takes the following form.

```
var Boxes
procedure M1M2
procedure G1
procedure G2
procedure P1
procedure P2
procedure Main
    loop
        P1
        G1
        M1M2
        G2
        P2
    end
Main
```

All type definitions are defined within the scope of P1/G1/M1M2/G2/P2. The solution then relies on two facts: 1. for each of the critical fields access is available only through access operations and 2. access to the access operations is controlled by access keys. If an access function is presented with a key which matches the key in the data structure to be accessed, then the access is allowed. Since keys cannot be forged, a routine could have the key only if it had been given the key through proper channels. There are two keys KeyM and KeyL; they correspond to being able to access MessageType and LetterType as in the Concurrent Pascal solution.

The main procedure starts by creating the two unique keys. All bundle and letter declarations subsequently are created to contain these keys which are handed out to P1/G1/M1M2/G2/P2 in accordance with required protection.

The keyword rep is used inside a cluster definition to denote the internal representation of the cluster.

Solution in Gypsy

Gypsy [1] is another language which is based on a Pascal syntax. It was specifically designed for supporting the development of formally verifiable programs for communications systems. Like Concurrent Pascal it offers facilities for writing programs using concurrent processes. It also provides facilities for writing program specifications directly in the program code. We will ignore the presence of these extras in the current context.

Gypsy programs are composed of a series of units, exactly one of which should be a program unit. Units are type, constant, macro, or routine units. Units may not be nested; hence, the traditional hierarchical structure is missing from Gypsy. Furthermore, variables can be declared only inside units. This means that there are only local and parameter variables.

```
MessageType = record[Body: String, Signature: PrisonerId];
LetterType = cluster is create, ReadAddress, ReadMessage, WriteLetter;
    rep = record[Key: KeyType, Address: PrisonerId, Message: MessageType];
    create = operation(k: KeyType) returns(rep);
        l: rep; l.Key:= k; return (l); end;
    ReadAddress = operation(l: rep) returns(PrisonerId);
        return (l.Address); end;
    ReadMessage = operation(l: rep, k: KeyType) returns(MessageType);
        if equal(l.Key, k) then return (l.Message); end;
    WriteLetter = operation(l: rep, a: PrisonerId, m: MessageType, k: KeyType) returns (rep);
        if equal(l.Key, k) then begin l.Address:= a; l.Message:= m; return (l); end; end;
    end;
BundleType = cluster is create, ReadLabel, WriteLabel,
        EmptyLetters, RemoveLetter, AppendLetter;
    rep = record [Key: KeyType, Label: PrisonerId, Letters: sequence of LetterType];
    create = operation(k: KeyType) returns(rep);
        b: rep; b.Key:= k; return (b); end;
    ReadLabel = operation(b: rep) returns(PrisonerId);
        return (b.Label); end;
    WriteLabel = operation(b: rep, a: PrisonerId, k: KeyType) returns(rep);
        if equal(b.Key, k) then begin b.Label:= a; return (b); end; end;
    EmptyLetters = operation(b: rep, k: KeyType) returns(boolean);
        if equal(b.Key, k) then return (empty(b.Letters)); end;
    RemoveLetter = operation(b: rep, k: KeyType)  returns(rep, LetterType):
        l: LetterType;
        if equal(b.Key, k) then begin remove l from b.Letters; return (b, l);
        end; end;
    WriteLetter = operation(b: rep, l: LetterType, k: KeyType) returns(rep);
        if equal(b.Key, k) then begin append l to b.Letters; return (b);
        end; end;
    end;
Mainbox = sequence of BundleType;
Mailboxes = array of BundleType;
M1M2 = procedure(masterbox: Mainbox, k: KeyType) returns(Mainbox);
        ...
    end M1M2;
G1 = procedure(boxes: Mailboxes, masterbox: Mainbox) returns(Mailboxes, Mainbox);
        ...
    end G1;
G2 = procedure(boxes: Mailboxes, masterbox: Mainbox) returns(Mailboxes, Mainbox);
        ...
    end G2;
P1 = procedure(box: BundleType, kb, k1: KeyType) returns(BundleType);
        ...
    end P1;
P2 = procedure(box: BundleType, kb, k1: KeyType) returns(BundleType);
        ...
    end P2;
Main = procedure
    KeyM: KeyType();
    KeyL: KeyType();
    Boxes: Mailboxes:= fill(1, NPrisoners, BundleType$create(KeyL));
    $fill creates an array indexed 1..NPrisoners;
    each element is initialized by calling the create operation on BundleTypet
    MasterBox: Mainbox;
    loop
        for i:= PrisonerId do Boxes[i]:= P1(Boxes[i], KeyL, KeyM);
        Boxes, MasterBox:= G1(Boxes, MasterBox);
        MasterBox:= M1M2(MasterBox, KeyL);
        Boxes, MasterBox:= G2(Boxes, MasterBox);
        for i:= PrisonerId do Boxes[i]:= P2(Boxes[i], KeyL, KeyM);
    end; end;
Main
```

Figure 11. Solution in Clu

```
type <A, B> IntStack <New, Push, Pop, Top> =
    record(st: array [1..100] of integer;
           pt: integer);
procedure <A, B> new: IntStack;
    begin result.pt:=0; end;
procedure <A> Push(s: IntSTack; i: integer) =
    begin s.pt:= s.pt+1; s.st[s.pt]:= i; end;
procedure <A> Pop(s: IntStack) =
    begin s.pt:= s.pt-1; end;
function <A, B> Top(s: IntStack): integer =
    begin result:=s.st[s.pt]; end;
```

Figure 12. Gypsy Access Lists

Associated with each unit are two access lists. The first (which appears syntactically before the unit name) lists those units that are allowed to access the unit. If this is a routine unit the list indicates which units are allowed to invoke the routine. For type units it indicates in which units objects of that type can be declared. When the first access list is missing, it is assumed that all units are allowed to reference the unit. The second access list (which is only meaningful for types and which syntactically immediately follows the unit name) indicates those units that are allowed to reference the internal structure of the unit. This access list allows for the construction of abstract data types. When this list is missing it is assumed that all units are allowed access to the internal structure. Figure 11 demonstrates the IntStack example again (Figure 7) - this time in Gypsy. The access lists state that both units A and B are allowed to declare an IntStack with Push, Pop, and Top being the only routines allowed to reference the internal representation of an IntStack. Furthermore, only A can do Push or Pop operations, but both A and B are allowed to do the Top operation. It should be clear that by supplying suitable access lists any graph structured access pattern can be achieved.

A type in Gypsy consists of a mode and a possibly empty set of restrictions. For instance, a subrange 1..10 (or integer[1..10]) is of mode integer with a range restriction requiring that the value be greater than or equal to one and less than or equal to ten. Two types "match" if their modes "match" after repeatedly substituting modes for their corresponding types. However, type impersonation is prevented by restricting objects from gaining access rights either through assignment or parameter passing.

Gypsy does not allow routines to be passed as parameters.

The Gypsy solution utilizes the following form.

```
procedure P1
procedure G1
procedure M1M2
procedure G2
procedure P2

program Main
    var Boxes
    loop
        P1
        G1
        M1M2
        G2
        P2
    end
end
```

The form of the solution itself requires little explanation. It requires no subtleties, but simply relies on the properties of its access lists. Several points are worth noting: 1. WriteLabel doesn't require a dummy parameter of type LetterType as some previous solutions have and 2. we are able to discriminate access to a finer level without increasing the complexity of the solution. Examples of the latter are WriteLetter being restricted to just P1 and P1/G1/M1M2/G2/P2 being restricted so that only Main is allowed to invoke them.

Conclusion

Is this example contrived? Yes and no. The PMS example was concocted to compress as much as possible complexity into a small example and at the same time to motivate as much as possible the significant protection problems. Hence, the PMS is contrived, but the protection problems are not.

Most of the protection problems illustrated here occurred to us in the process of constructing formal proofs for message communication systems. For the purpose of proofs it is not sufficient to rely upon programmer discipline. Hence, either the language guarantees tightly protected routine interactions or the verification system is forced to expand the scope of its proof process to account for possible side-effects that should never be allowed, but that the programmer has no facility of preventing. Routine specifications can be used to prohibit undesired routine interactions, but these "no-effect" type specifications tend to become voluminous and each instance still requires proof. We have found that by providing stronger access control facilities in the language the burdens of proof can be significantly reduced.

```
type MessageType = record(
    Body: String;
    Signature: PrisonerId);
type LetterType<ReadAddress, ReadMessage, WriteLetter> = record(
    Address: PrisonerId;
    Message: MessageType);
function <P1, M1M2, P2> ReadAddress(l: LetterType): PrisonerId =
    begin result:= l.Address; end;
function<P1,P2> ReadMessage(l: LetterType): MessageType =
    begin result:= l.Message; end;
procedure <P1> WriteLetter(var l: LetterType; a: PrisonerId; m: MessageType) =
    begin l.Address:= a; l.Message:= m; end;
type BundleType<ReadLabel, WriteLabel, EmptyLetters, RemoveLetter, AppendLetter> = record(
    Label: PrisonerId;
    Letters: sequence of LetterType);
function  ReadLabel(b: BundleType): PrisonerId =
    begin result:= b.Label; end;
procedure <P1, M1M2> WriteLabel(var b: BundleType; a: PrisonerId) =
    begin b.Label:= a; end;
function <P1, M1M2, P2> EmptyLetters(b: BundleType): boolean =
    begin result:= empty(b.Letters); end;
procedure <P1, M1M2, P2> RemoveLetter(var b: BundleType; var l: LetterType) =
    begin remove l from b.Letters; end;
procedure <P1, M1M2, P2> AppendLetter(var b: BundleType; l: LetterType) =
    begin append l to b.Letters; end;
type Mailboxes = array (PrisonerId) of BundleType;
type Mainbox = sequence of BundleType;
procedure <Main> P1(var b: BundleType) =
        ...
    end P1;
procedure <Main> G1(var boxes: Mailboxes; var masterbox: Mainbox) =
        ...
    end G1;
procedure <Main> M1M2(var masterbox: Mainbox) =
        ...
    end M1M2;
procedure <Main> G2(var boxes: Mailboxes; var masterbox: Mainbox) =
        ...
    end G2;
procedure <Main> P2(var b: BundleType) =
        ...
    end P2;
program Main =
    begin
        var Boxes: Mailboxes;
        var MasterBox: Mainbox;
        loop
            for i:= PrisonerId do P1(Boxes[i]):
            G1(Boxes, MasterBox);
            M1M2(MasterBox);
            G2(Boxes, MasterBox);
            for i:= PrisonerId do P2(Boxes[i]);
        end;
    end;
```

Figure 13. Solution in Gypsy

We again emphasize that in all fairness most of these languages were not designed to solve this type of a problem. We don't pretend that they were nor are we critical of their performance. Instead, our point is to illustrate how "existing" languages performed on the subject of protection, and indeed, they perform surprisingly well.

The solutions demonstrate several points about protection. First, never pass direct access to a protected object, but rather pass access to operations on that object. This point is the basis for the Pascal solution and the usage of abstract data types. Second, selective hiding of declarations is essential to a good solution. In Pascal, the declarations were hidden in an encompassed scope; in Concurrent Pascal, they were hidden by decoy definitions; in Euclid, they were hidden by selective exportation from modules; and in Gypsy, they were hidden by explicit access lists. Third, selective access to operations as well as data is important. In Concurrent Pascal and Clu, this was accomplished by constructing an unforgeable key; while, in Gypsy, we used an explicit access list. In Euclid, this was accomplished to some extent by nested definitions.

There are a couple of other works relevant to this problem. The design of a capability based protection scheme [6] appears to provide an elegant solution, as does Alphard [12]. While we examined both of these, we did not include solutions because of the lack of adequate language details required to construct a solution. Nevertheless, there is hope that in the near future even better capabilities for protection in programming languages may exist.

Bibliography

[1] Ambler, Allen L., Good, Donald I., and Burger, Wilhelm F. Report on the Language Gypsy. Technical Report ICSCA-CMP-1, Univ. of Texas, (1976).

[2] Brinch Hansen, Per. The Purpose of Concurrent Pascal. Proceedings ICRS (1975).

[3] Dijkstra, E.W. Notes on Structured Programming. Structured Programming, Academic Press, (1972).

[4] Hoare, C.A.R. Hierarchical Program Structures. Structured Programming, Academic Press, (1972).

[5] Jensen, Kathleen, and Wirth, Niklaus. Pascal User Manual and Report. Springer Verlag (1974).

[6] Jones, A. K., and Liskov, B. H. A Language Extension for Controlling Access to Shared Data. International Conference on Software Engineering (1976).

[7] Lampson, B.W. et al. Euclid Report. Xerox Research Center, Palo Alto (1976).

[8] Liskov, Barbara, and Zilles, Stephen. An Approach to Abstraction. Computation Structures Group Memo 88, MIT (1973).

[9] Palme, Jacob. New Feature for Module Protection in Simula. SIGPLAN Notices, 11, 5, (1976).

[10] Parnas, D. L. A Technique for Software Module Specification with Examples. Comm. ACM 15, 5 (1972).

[11] Wirth, N. Program Development by Stepwise Refinement. Comm. ACM, 14, 4, (1971).

[12] Wulf, W.A., London, R. L., and Shaw, Mary. Abstraction and Verification in Alphard: Introduction to Language and Methodology. Research Report ISI/RR-76-46, ARPA (1976).