ABSTRACTION IN THE INTEL iAPX-432 PROTOTYPE SYSTEMS IMPLEMENTATION LANGUAGE

Bruce J. MacLennan

April 1983

Approved for public release; distribution unlimited

Prepared for:
Chief of Naval Research
Arlington, VA 22217
The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:

BRUCE J. MACLENNAN
Associate Professor of Computer Science

Reviewed by:  
DAVID K. HSIAO, Chairman
Department of Computer Science

Released by:  
WILLIAM M. TOLLES
Dean of Research
Abstraction in the Intel iAPX-432 Prototype Systems Implementation Language

Abstract

This report describes the abstraction mechanism of a prototype systems implementation language for Intel's iAPX-432 microprocessor. Full exploitation of the 432's facilities places many demands on a language intended for systems implementation. The 432 is a capability-based machine, with hardware-enforced typing of large objects, dynamically instantiated domains (i.e., packages), hardware-enforced information hiding (seals), and hardware-supported, software-defined access rights (trademarks). The prototype language's support for these facilities is described in this project.
ABSTRACTION IN THE INTEL iAPX-432
PROTOTYPE SYSTEMS IMPLEMENTATION LANGUAGE

B. J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

1. INTRODUCTION

This report describes the abstraction mechanism of a prototype systems implementation languages for Intel's iAPX-432 microprocessor. The language was designed in 1977 by Bill Brown and myself (at Intel) and was implemented in Simula in 1978 and 1979. Intel has kindly declared this work non-proprietary, so its publication is now possible [Brown83]. The introduction to the language specification [PSIL78] describes the project's goals:

1. "To provide an adequate tool for programming the [iAPX-432]."
2. "To provide experience in the implementation of languages and systems for the [iAPX-432]."
3. "To provide a first cut at addressing the philosophical language design issues associated with concurrency, modularity, and protection.

"The prototype language is explicitly designed as a learning tool to establish the real requirements for meeting the above goals."

Although the prototype language is now five years old, I think that it has a number of unique characteristics that justify its description. Full exploitation of the 432's facilities places many demands on a language intended for systems implementation. The 432 is a capability-based machine, with hardware-enforced typing of 'large' objects, dynamically instantiated domains (i.e., packages), hardware-enforced information hid-
ing (seals), and hardware-supported, software-defined access-rights (trademarks). The prototype language's support for these facilities is described below. The 432 also provides a very dynamic, message-based model of concurrent execution; prototype language facilities to support this model are described in a companion report [MacL89].

The rest of this report essentially reproduces Section 3.1 and Chapter 4 of the prototype language specification [PSIL78]. To place this material in context it should be sufficient to know that the prototype language is an extensible data-abstraction language in the tradition of Alphard, CLU and MESA. However, to meet the requirements of the 432, it is generally more dynamic than these languages.

2. VALUES AND OBJECTS

Natural languages distinguish between common nouns and proper nouns. Proper nouns (or names) denote specific entities that exist (presumably). Common nouns denote concepts or abstractions, i.e., classes of entities, or classes of classes, etc. Abstractions and entities are compared and contrasted below.

Both entities and abstractions have attributes. For instance, if 'Caesar' is a name for a specific entity, we can speak of various attributes of this entity, such as the age of Caesar or the father of Caesar. Similarly, if the word $z$ refers to the complex number $1+2i$ (which is an abstraction), then we can speak of various attributes of this abstraction, such as the real part of $z$, or the imaginary part of $z$.

Abstractions and entities can be contrasted as follows. Entities are things that exist: as such, they can come into existence or go out of existence. They have attributes that can be changed in time without altering the basic identity of the entity. That is, an entity remains that same entity even though any or all of its attributes may have been changed. This includes the 'internal attributes,' or state, of the entity. Since entities have an identity which is distinct from the attributes possessed at any given point in time, it is possible that there can be two entities which have the same
attributes, yet are different entities. Such entities are called different instances of each other.

The concept of existence is not applicable to abstractions. Abstractions are timeless. i.e., it is meaningless to speak of them coming into existence or going out of existence. Since an abstraction is completely defined by its attributes, changing its attributes causes it to be a different abstraction. In this sense abstractions are unmodifiable. (It is, of course, possible to redefine the name of an abstraction. For instance, the word 'pi' might be redefined to refer to the abstraction 17, but this alteration does not alter that number which is the ratio of a circle's circumference to its diameter.) The fact that an abstraction is completely determined by its attributes also implies that the concepts of identity and instance are not applicable to abstractions.

Like natural languages, the prototype language distinguishes between entities, which it calls objects, and abstractions, which it calls values. The programmer generally deals with values (such as numbers or characters), except where updating, state information, or sharing are involved, in which cases objects are required. The naming of objects and values is discussed in Section 3.

3. SPECIFICATIONS AND BINDINGS

As was discussed in Section 2, the prototype language is capable of describing both values (abstractions) and objects (entities). To facilitate such description, values and objects can be denoted by words (or names). These correspond to the common and proper nouns of natural languages. This chapter describes how these words are defined, a process called binding. Values can also be described by 'denotations,' which are self-defining names for values. For example, '2' is a denotation for 2; it does not have to be explicitly defined. This chapter discusses the denotations for non-primitive values.
It has been shown that both objects and values have attributes. These attributes are usually named, but can be denoted by indexes, as is the case with arrays. (Ultimately all names are considered attribute names, since the names of variables, procedures, etc., are attributes of the environment.) This chapter discusses the ways in which names are associated with values and objects ('binding'), the ways in which one can restrict the class of values or objects to which a name will later be bound ('specification'), the ways of specifying classes of values and objects ('types'), the ways in which values can be constructed from more primitive values and objects ('composite' values), and the rules governing the context in which names are known ('scoping').

```
specification: bind-mode spec.
spec: [name] [id-list : type procedural-spec].
binding: [bind-mode bound-part].
bound-part: [name] [id-list [: type] = exp procedural-binding].
bind-mode: redefine nonref volatility.
volatility: [const var].
sp: specification ;
bd: binding ;
```

Figure 1. Specification and Binding Syntax

**Specification:**
```
var x: real;
proc fac(n: int) -> int;
```

**Binding:**
```
var x: real = 0;
pI = 3.14159;
proc fac(n: int) -> int is
if n=0 then return 1;
ext return n*fac(n-1);
end if;
end fac:
```

Figure 2. Specification and Binding Examples

The concept of a binding is of central importance in the prototype language. A bind-
ing is the formalization of the natural-language process of defining a word or name. In this process a common noun is associated with a particular concept, or a proper name is associated with a particular entity. In the same way a binding associates a name with a particular value or object (the language does not distinguish between common nouns and proper names). The name is said to be bound to the value of object. For instance,

\[ \text{const pi: real = 3.14159;} \]

binds the name 'pi' to the value denoted '3.14159.' The binding can be paraphrased "pi is defined to be the real number 3.14159." The word \textit{const} means that this definition is constant, or permanent, within the scope of the definition.

It is often useful to have a name that at various times can refer to different members of a class of values or objects. An example of such a 'variable' binding is:

\[ \text{var x: real = 3.14159;} \]

This could be paraphrased "x currently stands for the real number 3.14159." The binding is variable because the name 'x' can be rebound to another value of the same type (i.e., real) anywhere within the scope of 'x.' This is accomplished with an assignment operation. Formally, variables are just changeable attributes of a \textit{form object} (Section 5) representing the current environment. As a matter of convenience, the type can be omitted when it can be deduced from the bound value. Also, \textit{const} is assumed if it is omitted.

For the following discussion an understanding of AlgoL scope rules will suffice. It will usually be the case, as in AlgoL, that the current environment of known names is composed of those defined in the current (local) program unit together with those contained in outer (non-local) program units. In AlgoL, if the current program unit defines a name that already is defined in the non-local environment, then the new name supersedes the old. Such implicit redefinition is illegal in the prototype language, since it is a frequent source of errors. An name can be redefined in an inner scope, but the
programmer must make his intention explicit, by writing redefine. For example:

```let
t
```

In the prototype language, all bindings established within a given scope are interpreted to be mutually recursive. This means that the bodies on the right of the bindings 'see' the names on the left. This allows simply recursive functions to be defined in the obvious way, e.g.,

```proo fac(n:nt) = (n=0 => 1 | n*fac(n-1));```

This rule also allows sets of mutually recursive procedures to be defined, e.g.,

```proo f = ... g ... ;
proo g = ... f ... ;```

Sometimes it is useful to redefine a name in terms of its previous (more global) meaning. For this purpose the mutually recursive interpretation can be suppressed by writing nonrec. This means that the right-hand-side of the binding will 'see' only the non-local environment. For instance, if it were desired to redefine 'Sin' so that it worked in terms of radians rather than degrees, this could be done by:

```nonrec proo Sin(theta:real) = Sin(theta/180*pi);```

A binding defines the name on the left to be the current value or object described by the expression on its right. Thus, the binding 'const w = Sam.car.weight;' can be paraphrased "define w to be the current weight of Sam's car." The fact that the car's weight may later change will not affect the value of w. Occasionally it is desirable to introduce a name to stand for an attribute's value at all times. Thus, it might be desirable to define 'cw' to mean the weight of Sam's car, at any time. This can be done with
the binding:

\[
\text{Name } cw = \text{Sam.car.weight};
\]

This is an example of a 'name definition.' After this definition, 'cw' can be used anywhere 'Sam.car.weight' could have been used. For example, the weight of the car can be changed by 'cw := 4015:'.

A specification is essentially a binding without an initial value. It is used to restrict the set of values to which the name will be later bound (say by extension). Specifications usually occur in class-denotations (section 4). Examples of specifications will be found throughout this report.

4. TYPES

The concept of a type in the prototype language is very similar to a Pascal type or an Algol 68 mode. The differences will be discussed later. The type denotations (type-den) are the primitives which, with the type operators, are used to construct type-expressions. Throughout this document, the non-terminal type is used to denote such a type-expression. As in Pascal and Algol 68 a type denotes a set of values or objects that share certain attributes and operators. The specific sets are described below.

Perhaps the most familiar type denotation is the record-type denotation. A record (n-tuple, structure) denotes a unordered heterogeneous data structure. See the exam-
record-type-dev:
record re: real; ix: real; end

union-type-dev:
union l: int; R: real; end

enum-type-dev:
enum {maso, femn, neut}
enum {violet, indigo, blue, green,
yellow, orange, red}

class-dev:
class proc more -> Boolean:
  proc reset;
  proc next -> char;
end

Figure 4. Examples of Types

In Figure 4, Records in the prototype language provide facilities now quite common, such as initial (default) values for fields and position-independent initialization of fields. These facilities are justified and described in [MacL75], Chapter 6.

Since there are no 'references' in the prototype language, records can be directly recursive in definition. For example, the following is a definition of LISP-style lists:

cell = union atom: string;
 nonnull: list;
  null: {};
end

list = record car: cell, cdr: cell; end

If L is of type cell, then we can discriminate its variants by expressions like 'L is atom' or by a variant case statement (see [Hoare73]).

In natural languages, a class (concept, abstraction) is defined by stating the genus to which the members of the class belong and the attributes, attribute ranges or attribute values that distinguish the members of the class from the other members of the genus. This method of definition is captured by the class construct in the prototype language. Readers acquainted with the Simula or Smalltalk class should be on familiar
ground. Consider the class binding

\[ n = \text{class } g \text{ with } d \text{ end; } \]

The class being defined is 'n,' the genus is \( g \) and the differentia are \( d \). The binding can be paraphrased "define 'n' to be the class of all \( g \) such that \( d \)." The effect of the definition is to attach a name to all values or objects which are in the genus and satisfy the differentia (which are specifications). Each specification associates a set of possible values with an attribute name. If the attribute already exists as an attribute of the genus, then the respecification must be compatible with the old specification, i.e., the new set of values must be compatible with (i.e., be a subset of) the old set. An attribute is required to have a particular value by specifying a singleton set of value.

An example may clarify these ideas. Suppose class 'animal' had already been defined. The following additional classes are defined:

\[ \text{bird} = \text{class } \text{animal with wingspan: int; end; } \]

\[ \text{parrot} = \text{class } \text{bird with } \]

\[ \text{color: \text{enum } \{ \text{green, blue, grey, brown, mixed}\}; } \]

\[ \text{name: \text{string}; } \]

\[ \text{end; } \]

\[ \text{green_parrot} = \text{class } \text{parrot with } \]

\[ \text{color: \{green\}; } \]

\[ \text{end; } \]

\[ \text{large_parrot} = \text{class } \text{parrot with } \]

\[ \text{wingspan: \{50 to 1000\}; } \]

\[ \text{end; } \]

These bindings define a hierarchy of abstractions, each being a refinement of a preceding abstraction. Thus, a 'bird' is defined to be any animal with a wing span, a
parrot is defined to be a bird with one of the specified colors and a name, a green parrot is defined to be a parrot with color green, and a large parrot is defined to be a parrot with a wingspan greater than 50 cm.

A more useful class than parrots is defined by the binding:

file = class
  proc reset;
  proc more -> Boolean;
  proc next -> char;
  proc put (c:char);
  end file;

This defines a 'file' to be any object or value that has 'reset,' 'more' 'next' and 'put' attributes as specified. A procedure to copy one file to another could be defined:

proc copy (f1:file, (* to *) f2:file) is
  f1.reset;
  while f1.more repeat
    f2.put (f1.next);
  end
end copy;

This procedure will work on any values or objects that have the specified attributes. For instance, they might be disk or tape files or arrays or sequences of characters in memory.

Sometimes the only attributes two or more types share is the fact that they participate in a collection of operations or relations. To allow this the prototype language provides for the denotation of types which are the discriminated union of other types. (See the preceding definition of 'cell.')
forms-dsn: form [extension] form-body end [form].
extension: exp with.

Figure 5. Syntax of Forms

5. FORMS

Forms provide a mechanism for directly constructing values by defining their attributes in terms of other values and objects. A form is a collection of bindings, which comprise the attributes of the value. The attributes may be procedural, data, type, or other values or objects. Unlike classes, the attributes of a form are divided into two groups, the private attributes and the public attributes. The public attributes are signified by the word public preceding the bindings. These attributes can be made visible outside the form through the with statement (described later). The names and types of the public attributes determine the type of the form.

An object can be constructed according to a form by preceding the form with obj. This is the primary mechanism for directly constructing objects from other values and objects. Examples will be seen below.

One common use of form values is to define 'libraries' of related procedures, constants and types. For instance, a library for complex arithmetic could be defined as in Figure 7. When such a library has been defined, it can be used as follows:

with CompArith do
...
let var x: complex;
let var a, b, c: complex;
...
if z = 1 then z := a * b / c; end;
...
end:
form
  public var x: real;
  public var y: real;
  public proc rho = (x2 + y2)^(1/2);
  public proc theta = arctan(y/x);
end

Figure 8. Example of Form
Since a library is just a set of bindings between names and objects or values, and as such has no 'memory' (i.e., state information) it is appropriate that it be defined as a form value (as opposed to a form object). An example of a structure which does have memory, and thus should be implemented as a form-object, is a stack. A particular message stack, 'Msgstk' can be defined by a binding such as that in Figure 8 (the sequence operations are built in and the type message is assumed to have been defined). It is now possible to push messages onto and pop messages off of Msgstk:

let var m,n: message;
...
Msgstk.push (m);
...
if not Msgstk.empty then n := Msgstk.pop; end

The combined powers of classes and forms provide a very useful facility, namely, the ability to have multiple implementations of a single abstract type. As an example, the abstract type 'message stack' will be defined. One form will use the sequence implementation used in the previous example, the other will use finite arrays. The abstract concept of a message stack is defined by the following class:

message_stack = msgt object
  where msgt = class
  proc push (m: message);
  proc pop -> message;
  proc empty -> Boolean;
end

-12-
CompArith = form
public complex = record re: real; im: real; end;
public 0const i = complex (0,1);
public proc 6 (x: complex) + (y: complex) =
complex (x.re + y.re, x.im + y.im);
public proc 6 (x: complex) - (y: complex) =
complex (x.re - y.re, x.im - y.im);
public proc 7 (x: complex) * (y: complex) =
complex (x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re);
end form;

Figure 7. Form for Complex Arithmetic
A procedure 'seq.mstack' (for 'sequence-type' message stack) is now defined which returns a new sequence-based stack object. The actual definition of these objects is the same as Magstic, see Figure 9.

An alternative implementation of 'message stack' is provided by the procedure 'arr.mstack' (for 'array-type' message stack) which returns a new array-based stack object of a given size. See Figure 10. Note that a form-returning procedure has been used to get the effect of 'generic' forms; unlike in Ada, a separate generic mechanism is not required in the prototype language. Note also that 'arr.mstack': have an additional attribute, 'full' which inquires whether the stack is full. This attribute makes no sense for 'seq.mstack's since they are unbounded in size. Regardless of this extra attribute, both 'seq.mstack's and 'arr.mstack's are of type 'message_stack.' This is because they both satisfy the definition of 'message_stack,' i.e., they have the required attributes with the given specifications.

The following program fragment declares several stacks using these procedures (including Magstic) and declares a 'stack variable,' CurrentStack, which at various times will refer to either sequence or array based stacks.
The last statement uses the `has` operation to determine if the stack now referred to by `CurrentStack` has a 'full' attribute.

The extension part of a form allows one form to be created which is an extension of another form. That is, a new form can be created by adding or respesifying attributes of an existing form, which is similar to the Simula and Smalltalk subclass mechanisms. It is here illustrated by an example adapted from the DEC-10 Simula manual. Consider a form that manipulates vectors (Figure 11). Note that the procedure 'norm' is not bound, it is only specified, even though it is used in the 'normalize' procedure.
```plaintext
proc seq_msgstack =
  obj form
  var st: message sequence = [];
  public proc push (m: message) is
    st := [m] + st; end;
  public proc pop is
    let top = st.first;
    st := st.first;
    return top;
    end;
  public proc empty = (st = []);
  end form;

Figure 9. Sequence-type Message Stacks

A specific 'norm' procedure can be bound in an extension of 'row.' To continue the example, two extensions of 'row,' with different 'norm' procedures, are defined; see Figure 12. Thus 'row1.normalize;' will normalize its array using the first 'norm' and 'row2normalize;' will normalize its array using the second 'norm.'

6. Attribute Composition

The attribute composition operators allow the manipulation of the attributes of values and objects. The expression

\[ z \text{ excluding} (id_1, id_2, \ldots, id_n) \]

is the same object or value as \( z \), except that the attributes \( id_1, id_2, \ldots, id_n \) are no longer available; they have essentially been made private. For instance, if it were desired to pass SymTab to a procedure \( P \) in such a way that \( P \) could not enter anything into SymTab, then an appropriate invocation would be:

\[ P(\text{SymTab excluding} (\text{enter})); \]

Sometimes it is easier to state the attributes that are to be kept than to state those that are to be deleted. This the purpose of the including operator. The expression:

\[ z \text{ including} (id_1, id_2, \ldots, id_n) \]

is the same object or value as \( z \), except that all attributes other than \( id_1, id_2, \ldots, id_n \) are no longer available; i.e., the only public attributes are \( id_1, id_2, \ldots, id_n \). For instance, if center is a two-dimensional position (with both Cartesian and polar
```
proc arr-mystack (size: int) =
  obj form
  var st: message array [1 to size];
  var t: [0 to size] = 0;
  public proc push (m: message) is
    if full then error; end:
    t := t + 1;
    st[t] := m;
  end:
  public proc pop -> message is
    if empty then error; end:
    t := t - 1;
    return st[t + 1];
  end:
  public proc empty = (t = 0);
  public proc full = (t = size);
  end form:

Figure 10. Array-type Message Stacks

coordinates), then a strictly polar version of the value is:

    center including (rho, theta)

The last attribute composition operator is merge. If x and y are objects or values, then x merge y is a value with all of the attributes of both x and y. More precisely, for every attribute of either x or y, there is an attribute in x merge y with the same name as that attribute in x or y. Of course, x merge y is defined only if the identifiers for the attributes of x and y are disjoint. The merge operator is usually used in conjunction with the with statement. For example, if Math_Lib and Plot_Lib are two forms containing libraries of procedures, then all the attributes of both can be made available by:

    with Math_Lib merge Plot_Lib do

    ...

    end with:

If the only procedures needed from Math_Lib are Sin and Cos, then the following would be better:

    with Plot_Lib
    merge Math_Lib including (Sin, Cos) do
row = obj form
    public var A: real array;
    public proc norm -> real;
    public proc normalize is
        let var t = norm;
        if t <> 0 then
            t := 1/t;
            for name ai in A repeat
                ai := ai * t;
            end
        end
        end normalize;
    end form;

Figure 11. Form Object to Manipulate Vectors

7. TRADEMARKS AND SEALS

7.1 Trademarks

As discussed in section 4, a set of named (or numbered) attributes and the set of values or objects to which they may be bound determine a class. In that section the class file was defined:

file = class
    proc reset;
    proc more -> Boolean;
    proc next -> char;
    proc put (a: char);
end class;

This defines a 'file' to be any object or value with attributes 'reset,' 'more,' 'next' and 'put' of the types specified. This is a powerful and flexible facility. It allows the definition of procedures such as Copy (defined in Section 4) that copies any 'file' to any other 'file.' There may be many implementations of files, e.g., disc-files, character sequences, and character generators, as long as they define the stated attributes. There is, of course, no guarantee that the attributes of a particular file implement the
Extensions of Row

functions implied by their English names; it is only required that the types match. This is sometimes unsatisfactory. In particular there will be circumstances in which a file (for example) is required which has been formally or informally verified to satisfy certain properties. For instance, we would expect that writing a file, resetting it, and then reading it would produce the original data. Since the prototype language includes no direct support for verification, some other means must be provided for this protection. This is the trademark. It is essentially the same as the transparent seal described in [Morris73].

Anyone can construct 'files.' The danger is that, although they must satisfy the class definition, the files may be defective in some subtle way (e.g., are write-only) or are otherwise unacceptable. In the real world the consumer can protect himself by obtaining his files from a 'reliable source,' i.e., a source that he is confident will provide him with an acceptable 'file.' In the real world there are two ways a consumer can ensure that a given 'file' comes from this reliable source:

1. Request it directly from the reliable source.

row1 = obj form row with
  public proc norm is
  let var t: real = 0;
  for ai in A repeat
    t := t + ai;
  end;
  return t + .5;
  end norm;
end form;

row2 = obj form row with
  public proc norm is
  let var s: real = 0;
  let var t: real;
  for ai in A repeat
    t := abs ai;
    if t>s then s:=t; end;
  end;
  return s;
  end norm;
end form;
2. Require that it bear the ‘trademark’ of the reliable source.

Case (1) is straight-forward and requires no further discussion. The trademark which an object or value bears is an attribute, just as, for instance, that object’s or value’s color. The difference is that the generation and attaching of trademarks is strictly controlled. In the real world this is a function of the government (since a trademark is private property); in a computer system it is administered by the programming language and enforced by the operating-system and hardware.

In the prototype language, trademarks are declared only in forms and classes. Such a declaration takes the form:

```
trademark Acme;
```

which declares the trademark ‘Acme.’ This has two effects: within the form in which the declaration appears, an expression such as \( x \text{ qua } \text{Acme} \) returns a version of \( x \) with the trademark \( \text{Acme} \). Outside of the form of declaration the trademark’s name can be seen (like other publics of the form), but not used for applying trademarks. An expression such as

```
if y is Acme then ...
```

will determine whether \( y \) has the Acme trademark. A file bearing the Acme trademark is denoted by ‘Acme & file;’ using & is the type-intersection operator. Thus, if it were desired that Copy only work on Acme files, its procedure head could be written:

```
proc Copy ( f1: Acme & file, f2: Acme & file) is ...
```
Syntax.

```plaintext
label-binding: label-variety id-list .

label-variety: [trademark]

Examples.

trademark standard:

seal atom null:
```

Figure 14. Syntax for Trademarks and Seals

Of course it is possible to have more than one trademark on a value or object, or to use the same trademark on several classes of values or objects. (Acme may also make very fine stacks!)

The example in Figure 15, which allows the use of both degrees and radians, is a non-traditional use of trademarks (i.e., units). Note that we have also overloaded the assignment operation: this defines coercions between radians and degrees.

```

DoubleTrig = form

trademark deg:

trademark rad:

pi = 3.14159 2653589;

public type degrees = deg & real;

public type radians = rad & real;

public const right_angle = 90 qua degrees;

public nuoce proc Sin (Theta: degrees) = Sin (Theta);

public nuoce proc Sin (Theta: radians) = Sin (Theta * pi/180);

public proc (name Theta1: radians) := (Theta2: degrees) is

Theta1 := (Theta2 * pi/180) qua rad:

end;

 Theta1 := (Theta1: radians) is

Theta1 := (Theta1 * 180/pi) qua deg;

end;

public proc (Theta1: degrees) + (Theta2: degrees) =

((Theta1 + Theta2) \ 360) qua deg;

end;

```

Figure 15. Implementing Units with Trademarks

These declarations allow the use of angles measured in either radians or degrees.

Further, they ensure that the appropriate Sin routine is used for each unit.

7.2 Seals

The main purpose of a trademark is the protection of the user of a value or object.
This is accomplished by unforgeably identifying the source of a value or object to its potential users (which users may include the object's or value's creators). A related construct is the seal, which can loosely be described as a trademarked box [Morris73]. That is, the object's or value's originator is unambiguously identified as with a trademark, but all other attributes of the value or object are hidden outside of the form in which it is declared. That is, the object or value appears atomic outside the form in which the seal is declared. Inside this form the seal acts just like a trademark, i.e., all the attributes are visible. For example, the form in Figure 16 provides a collection of procedures for creating and manipulating 'particle' values. Outside the form, 'particles' are atomic.

```
Particle_Lib = form
  seal particle;
  part = record
    spin: {+1, -1};
    charge: [-3 to 3];
    strangeness: [+1, 0, -1];
    charm: [-1, 0, +1];
  end;

public u_quark = part (+1, +2, 0, 0) qua particle;
public d_quark = part (-1, -2, 0, 0) qua particle;
public s_quark = part (+1, -1, +1, 0) qua particle;
public c_quark = part (+1, +2, 0, -1) qua particle;
public pro charge (p: particle & part) -> resl = p.charge/3;
public proton = part (+1, +3, 0, 0) qua particle:
  ...
public proc (p: particle & part) + (q: particle & part)
  = part (p.spin + q.spin,
           p.charge + q.charge
           p.strangeness + q.strangeness,
           p.charm + q.charm) qua particle;
  ...
end form:
```

Figure 16. Example of Seals

It will then be possible to write statements such as:

```
with Particle_Lib do
  ...
  if proton = u_quark + u_quark + d_quark then ...
```

The 'quantum numbers' (such as spin and charge) are hidden outside the form except...
where explicitly made available (as is done with charge, above).

In summary, it can be seen that seals provide another level of security beyond trademarks. Seals, like trademarks, guarantee that only the owner of the seal can create the sealed objects or values. Seals enforce the further restriction that only the owner of the seal can inspect the attributes of the sealed objects or values.

3. VISIBILITY, OWNERSHIP AND EXTENSION

The prototype language distinguishes between the scope of a variable and the visibility of a variable. The scope of a binding is determined by the type of the binding and the static nesting of program components. Generally, a binding can be seen only within its scope, although there are circumstances in which it is visible outside its scope. For instance, the with construct provides access to the publics of a form; in other words, with makes the publics visible throughout the body of the with.

The environment in which a binding is made is defined to be the owner of that binding, and any object or value created in that environment is likewise owned by that environment. The owner of bindings, objects and values has special privileges not possessed by other environments to which the names, values and objects may be visible. These special privileges are, however, inherited by any environments in the scope of the bindings.

The above named privileges hinge around the ability to see the private bindings of a form. In particular, in the scope of a form creation the private bindings will be accessible just like the publics. This is especially important to the extension operation, since an extension to a form will 'see' the private bindings of that form only if the extension is made in the environment of the form's creation. An example may clarify these ideas. Recall the definition of 'seq_mstack' (sequence-based message stacks) in section 5. Assume that this is a public binding in some form F. Further, assume that someone not in environment F wants to extend seq_mstacks with a new operation, 'pushall,' such that
S.pushall [X1, X2, ..., Xn]

will push all of X1, X2, ..., Xn onto S. The with construct must be used to make the
name seq.mstack visible. The form denotation is then used to perform the extension.
Note, however, that since only the publics are visible in the extension, only they can be
used to implement 'pushall' (Figure 17).

with F do
  let proc multi_seq_mstack =
    obj form seq.mstack with
    public proc pushall (ms: message sequence) is
      for m in ms repeat
        push m:
      end for;
    end pushall;
  end form:

end with;

Figure 17. Extending a Form

If, however, the extension were made in the owning environment, F, then the private
bindings of the seq.mstack would be available, thus permitting a simpler implementa-
tion:

proc multi_seq_mstack =
  obj form seq.mstack with
  public proc pushall (ms: message sequence) is
    St := St + ms;
  end pushall;
end form:

In this case pushall is implemented by directly manipulating the private data-
structure, St.

9. REFERENCES


[Hoare72] Hoare, C.A.R., Recursive Data Structures, Stanford University Computer Sci-
ence Department STAN-CS-73-400; also Stanford Artificial Intelligence Laboratory
MEMO AIM-223; October 1973.


-24-
INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93940

Office of Research Administration 1
Code 012A
Naval Postgraduate School
Monterey, CA 93940

Chairman, Code 52Hq 40
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Professor Bruce J. MacLennan, Code 52MI 12
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Dr. Robert Grafton 1
Code 433
Office of Naval Research
800 N. Quinch
Arlington, VA 22217

Dr. David W. Mizell 1
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106

John M. Hosack 1
Department of Mathematics
Colby College
Waterville, ME 04901

Dr. David B. Lomet 1
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Jim Bowery 1
Viewdata Corporation
3rd Floor
1111 Lincoln Road
Miami Beach, FL 33139
J. Craig Cleaveland
1F35
Bell Laboratories
1600 Osgood Street
North Andover, MA 01845

Professor John M. Wozencraft, 62Wz
Department of Electrical Engineering
Naval Postgraduate School
Monterey, CA 93940

Mark Himmelstein
1323 Tulip Way
Livermore, CA 94550

Mr. William L. Brown
Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro, OR 97123

Mr. H. M. Gladney
IBM Research Laboratory
5600 Cottle Road
San Jose, CA 95193