CMU-CS-78-132

TARTAN

Language Design for the Ironman Requirement: Notes and Examples

Mary Shaw
Paul Hilfinger
Wm. A. Wulf

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

June, 1978

Abstract: The Tartan language was designed as an experiment to see whether the Ironman requirement for a common high-order programming language could be satisfied by an extremely simple language. The result, Tartan substantially meets the Ironman requirement. We believe it is substantially simpler than the four designs that were done in the first phase of the DOD-1 effort. The language definition appears in a companion report; this report provides a more expository discussion of some of the language's features, some examples of its use, and a discussion of some facilities that could enhance the basic design at relatively little cost.

This work was supported by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074 (monitored by the Air Force Office of Scientific Research).

Tartan: Notes and Examples

1. Notes on Important Issues

The Tartan reference manual is the defining document for the Tartan language. However, some of the facilities designed in response to the Ironman requirement deserve more unified and expository explanations than can be included in a reference manual. This chapter discusses the Tartan solutions to several important problems posed by the Ironman requirement.

The Tartan language draws heavily on the Pascal tradition. Both the reference manual and these notes assume familiarity of Pascal-like languages. These notes also assume familiarity with the Ironman requirements [1] and the Tartan reference manual [2].

1.1. Vocabulary

A Tartan program is made up of definitions, declarations, and (executable) statements. A definition binds an identifier to a module, routine (function, procedure, or process), type, or exception; it is processed during translation. A declaration binds an identifier to an object (i.e., a variable or value); it is processed at run time, usually to allocate storage. Executable statements are elaborated at run time to effect computations -- manipulation of values.

Identifiers can be bound to modules, routines, types, objects, statements, and exceptions. Individual identifiers are qualified with the names of the modules in which they are defined in order to avoid conflicts with names declared in other modules.

The computation described by a program is carried out by elaborating the program. We use the word "elaboration", in preference to "execution", to connote actions taken during translation as well as those taken during execution. Elaboration may be thought of as an idealized direct execution of the textual version of the program. The effect of elaborating each construct in the language is given in the reference manual.

Although the language prohibits making a declaration that gives new meaning to an identifier in a given scope, duplicate identifiers might arise in three situations. These situations, and the way Tartan deals with them, are:

- The same identifier is exported from two modules. The ambiguity is prevented by name qualification: All identifiers exported from a module are prefixed with the name of the module that exported them; the prefix is separated from the identifier by an apostrophe. Thus if identifier x is exported to the same scope by both modules M and N, we write

```
M'x                    ! for the x exported from M
N'x                    ! for the x exported from N
```

  The qualification may be omitted if no ambiguity arises.

- An identifier is used as an overloaded routine or type name. That is, the same routine name is given several definitions with different numbers or types of parameters. Operator overloading is permitted so that similar operations on distinct types, particularly infix operations, can be given the same names. The identifiers for the routines or types are disambiguated by examining the parameter types and choosing the routine whose formal parameter types are matched by the types of the actuals. A similar situation exists with identifiers for families of related types. In order to discuss these situations, we introduce the notion of signature:

  - The signature of a routine is the routine name together with its formal parameter types. The type of the value returned by a function is not included in its signature.

  - The signature of a type is its simple type name together with its generic characteristics. Generic characteristics are discussed in Section 1.5.1.

- A literal or constructor might potentially be of two or more different types. The ambiguity is resolved by qualifying the literal or constructor with the intended type, including its attributes.

1.2. Scope and Extent

Scope determines the interpretation of identifiers, so all the text in a given lexical scope shares the same vocabulary -- definitions, variables, etc. **Scope rules** permit some identifiers to be used with the same interpretation in multiple lexical scopes.

The **extent** of a variable is its lifetime -- the time during which it affects or is affected by the elaboration of the program. The interaction of control and lexical structure determines extent. Binding is the association of identifiers with program entities (objects, modules, routines, types, statements, and exceptions). The bindings in effect at any time result from the interaction of control and lexical scope.

1.2.1. Scope

Lexical structure is imposed on Tartan programs by **blocks** and **modules**, which delimit **lexical scopes**. There are no restrictions on the ways these scopes may be nested. Both constructs may use identifiers defined in other scopes; both may define identifiers that can be used in other scopes. Scope rules govern the legal bindings of identifiers in a lexical scope to program entities; they also control the **importing** and **exporting** of identifiers to other scopes. Blocks and modules differ only in their scope rules and in their effects on the extent of variables. Tartan has two scope rules:

- An **open scope** inherits (imports automatically) all the identifiers that are defined in its enclosing scope. It may not export any identifiers to its enclosing scope. Blocks are open scopes except when used as routine bodies.

- A **closed scope** inherits all identifiers that are defined in its enclosing scope except those for labels and objects.[1] It may explicitly import identifiers for objects (variables and constants), provided they have global extent. A closed scope that is a module may **export** identifiers that name variables, definitions, or exceptions; the exported identifiers have the status of any other identifiers defined in the enclosing scope. All modules are closed scopes, as are blocks when they are used as routine bodies.

Identifiers that are exported from an inner scope or imported from an outer scope have the status of identifiers defined in the scope. Redefinition of identifiers within a scope is not permitted. The convenience of being able to do so does not offset the danger of confusion. This does not, however, prohibit **overloading** of routines names; the differences in signatures suffice to prevent confusion. In addition, the same identifier might be imported with different meanings from two different modules; such identifiers are qualified with the names of the modules in which they were defined. Thus they are not duplicate definitions. Similarly, literals and constructors are qualified with their types to prevent ambiguity. In either case, the module or type qualifier may be omitted if no ambiguity arises.

1.2.2. Extent

Extent rules govern the lifetimes of objects. Extent is controlled by blocks, independent of whether they correspond to open or closed scopes. Nothing except blocks controls extent. The static data of a block is allocated when the declarations of the block are elaborated (in lexical order) at block entry. It is deallocated when the block is exited or terminated. Note that modules do not define extents, so the extent of data defined in a module coincides with the extent of its surrounding block.

Values of dynamic types point to dynamically allocated variables. The type of object that may be pointed at is part of a dynamic type. The extent of dynamically allocated variables is coincident with the blocks in which the associated dynamic types are declared. Since type names are not accessible outside the blocks in which they are defined, no references can outlive the block with which the extent is associated.

---

[1] Literals and identifiers for variables that are declared manifest are inherited.

1.3. Modules and Routines

Modules and routines are closed scopes. Modules serve as an encapsulation mechanism, protecting the privacy of definitions and declarations without restricting their extent. Routines are used for program structuring and abstraction of operators; they define operations that may be invoked during elaboration of a program.

1.3.1. Modules

A module is a closed scope that allows local definitions to be shared without making them public. It also serves to bundle up related definitions for administrative (program organization) purposes. It may export identifiers for definitions and objects to the scope in which it is defined. A module has no parameters.

A module is purely a scope-defining device. Its elaboration takes place during the elaboration of declarations for the block in which the module is defined. This elaboration consists of elaborating the definitions and declarations of the module in lexical order, then elaborating the statements of the module.

A module or routine inherits identifiers for definitions (modules, routines, types, and exceptions), literals, and manifest objects from its enclosing scope. It may explicitly import identifiers of objects from that scope, provided the objects have global extent. A module, but not a routine, may export identifiers other than labels to its enclosing scope.

1.3.2. Routines

A routine is a closed scope whose body is a block. Thus its body controls extent for local declarations, but does not inherit identifiers for variables or non-manifest constants. The <formals> list declares the identifiers for parameters.

A routine may be a function (func), which returns a value and has no visible side effects; it may be a procedure (proc), which can modify its parameters but must be called as a statement; or it may be a process, which is a potentially-parallel procedure. Special type-specific routines for many types are listed in the Tartan Reference Manual.

The symbols for the unary and binary operators are used as routine names in order to provide overloaded definitions for those operations.

If a <binding> in a routine header is omitted, it is assumed to be const. The result binding may be used only in <formals> lists of procedures. Functions are permitted to specify var parameters in order to avoid the copy associated with const.[1] However, as noted below, visible side effects on such parameters are prohibited. No duplication of identifiers within the <formals> list is permitted. Further, formal parameter names may not conflict with declarations or imports in the routine body.

If a routine name is overloaded, the definition whose signature matches the call is selected.

During elaboration of a function, assignment to a variable that is not local to the function body (or to the body of any routine it invokes, directly or indirectly) is permitted only if the function is never invoked in a scope where such a change is made to a variable or component that is directly accessible by the caller. Such variables may be imported by the function from a module within which the function is defined. They may also be fields of var parameters if the type of the parameter is defined in the same module as the function and the field name is not exported. An example of the latter case appears in section 2.4.

This is a compromise solution to the side-effect problem. Many routines are quite reasonably coded as value-returning: Get of section 2.4, monitoring routines, random number generators, and Pop for stacks. However, the current state of the art does not offer a sharp rule from distinguishing safe from unsafe side effects.

---

[1] In the presence of parallelism, it may not be safe to optimize away the copy of a const parameter even if the routine does not alter it.

Actual parameters are matched with formal parameters positionally. They must satisfy restrictions on type, binding and aliasing.

- The type of an actual parameter is acceptable if its <type name> exactly matches the <type name> of the corresponding formal parameter. Type attributes (instantiation parameters of a type) play no role in type checking.

- The binding of the actual parameter is acceptable if it matches the <binding> of the corresponding formal parameter according to the following rules:

If the formal parameter is	then the actual parameter may be
var	<var id>
const	<expr>
manifest	any manifest <expr>
result	<var id>

- Finally, the set of actual parameters must satisfy the following nonaliasing restriction: A variable may not be used in more than one var or result position of a single procedure or process call. For the purpose of testing this restriction, imported variables are considered to be var parameters.

1.4. Generic Definitions

A facility for making **generic definitions** is provided in order to allow the programmer to write a single textual definition that serves as an abbreviation for many closely-related specific definitions. Modules and routines may be defined generically.

A generic definition is instantiated by referring to it as the body of a module or routine definition. The effect of the instantiation is as if the generic definition were lexically substituted in place of the reference to it. That is, the body of the module or routine being defined becomes a copy of the generic definition.

1.4.1. Writing and Using Generic Definitions

A generic definition is syntactically like the corresponding specific definition except that it is prefixed by the word **generic** and it may have a set of generic parameters (enclosed in square brackets) after the name of the construct being defined. The parameters may be any defined identifiers, including those for variables, routines, types, or modules, or any expression. When the generic definition is instantiated, the text of the actual parameters replaces the identifiers that represent the formal parameters. The substitution is done on a lexical, rather that a strictly textual, basis. That is, the identifiers in the generic definition are renamed as necessary to avoid conflicts with the identifiers in the actual parameters.

For example, the collection of functions

```
func F2(X:Int)y:Int; begin y := 2 * X end
func F3(X:Int)y:Int; begin y := 3 * X end
func F4(X:Int)y:Int; begin y := 4 * X end
   and so on
```

can be defined by the generic definition

```
generic func F[Mult:Int] (X:Int)y:Int; begin y := Mult * X end
```

and the specific instantiations

```
func F2 is F[2]
func F3 is F[3]
func F4 is F[4]
   and so on
```

An instantiation of a generic definition may be used as the body of a specific module or routine. The usual restrictions on defining new identifiers apply to the module or routine being defined in terms of a generic.

Generic type definitions arise from generic modules. They are instantiated when the module is instantiated. Thereafter, they may be used in declarations or definitions.

If the generic definition has generic parameters, the actual parameters supplied with the instantiation must have correponding types and be syntactically suitable for substitution.

If a generic definition is instantiated more than once in a scope, ambiguous names may be introduced. The usual rules for resolving such ambiguities apply.

1.4.2. Separate Definitions

Tartan supports separate definitions, and potentially separate compilation, in the same way as it supports generic definitions. A program may be broken into separately defined segments. This decomposition must take place in the global extent. The units of separate definition are modules and routines. The definition

```
module Q is assumed(I)
```

in a segment has the effect of making the semantics of the segment the same as if the (separately defined) text of Q had been substituted for "is assumed(I)". The identifier I refers to a file, library, or other facility for storing separately defined segments. The relation between the identifier I and that storage facility may be established by a pragmat.

Suppose we want to develop and maintain a program with the following structure:

```
begin
module COM; begin export X; . . . end;
module M1; begin import X,Y; export Z; . . . end;
module M2;
    begin import X,Z;
    export W;
    module M3; begin . . . end;
     . . .
    end;
var Y: . . .;
! Main program using W, X, Y, Z
end;
```

If the definitions of COM, M1, and M2 are stored in a library, the following program will have the same effect:

```
begin
prag Require(ComDef,LIB.COM.TXT); Require(M1Def,LIB.M1.REL);
   Require(M2Def,LIB.M2.REL); garp;
module COM is assumed (COMDef);
module M1 is assumed (M1Def);
module M2 is assumed (M2Def);
var Y: . . .;
! Main program using W, X, Y, Z
end;
```

We assume here that the second argument of the Require pragmat is interpreted by the system as a pointer into a library. From the standpoint of the language, it is a matter of optimization whether the separate definitions are included as text or separately translated and linked in.

In order to perform independent translations of a separately defined module, it is necessary to embed it in an environment that provides the definitions it depends on. This environment must form a complete program. The translation system is assumed to provide commands for selecting which components of such a translation to save and for determining where and in what form they are saved. In the examples here, we will simulate that facility with a pragmat located outside the program. In the example above, module COM does not depend on any external definitions. In order to compile it separately, we write simply:

```
prag Save(Com,LIB.COM.TXT); garp;
begin
module COM; begin export X; . . . end;
end
```

Module M1 depends on the X exported from COM and the Y declared in the main program. To translate M1 separately, we must therefore write:

```
prag Save(M1,LIB.M1.REL); garp;
begin
prag Require(ComDef,LIB.COM.TXT); garp;
module COM is assumed (COMDef);
module M1; begin import X,Y; export Z; . . . end;
var Y: . . .;
end
```

If module M2 were translated monolithically, its translation environment would look much the same. Suppose, however, that the definitions of M2 and M3 are to be separated. They can be translated independently with the following two programs:

```
prag Save(M2,LIB.M2.REL); garp;
begin
prag Require(ComDef,LIB.COM.TXT); Require(M1Def,LIB.M1.REL);
   Require(M3Def,LIB.M3.REL); garp;
module COM is assumed (COMDef);
module M1 is assumed (M1Def);
module M2;
    begin import X,Z;
    export W;
    module M3 is assumed(M3Def);
     . . .
    end;
end
```

```
prag Save(M3,LIB.M3.REL); garp;
begin
prag Require(ComDef,LIB.COM.TXT); Require(M1Def,LIB.M1.REL); garp
module COM is assumed (COMDef);
module M1 is assumed (M1Def);
module M2;
    begin import X,Z;
    export W;
    ! Only the declarations of M2 that are required by M3 appear
    module M3; begin . . . end;
    end;
end
```

1.5. Types

The notion of type is introduced into languages to govern the ways operations are applied to objects. Types determine certain properties of data (values), including what operations on the values are legal and precisely what their effects are. Every object has a fixed type. This type is determinable during translation. The <type name> is determined by the signature of the type as described in section 1.5.2. Tartan provides certain built-in types; these include both simple and composite types. The user may define new types on the basis of these primitives. Both user-defined and built-in types are used to ensure that the actual parameters passed to a routine match the corresponding formal parameters. The types of the formal parameters are also used to construct the signature of a routine in order to resolve overloading ambiguities.

In Tartan, every value has exactly one type. This type is determined

- by the declaration of a variable or definition of a function
- by the lexical form and context of occurrence of a literal

Types appear in four contexts:

- in declarations, to give the type and attributes of an object
- in type definitions, to give the base representation of a newly-defined type
- in formal parameter lists, to restrict the objects that may be passed as parameters
- in function definitions, to give the type of the result

1.5.1. Characteristics and Attributes

Some of the properties of a type are the same for all values and objects of the type. These are called **generic characteristics** and are discussed below. Other properties of a type, called **attributes**, may differ from one value or object of the type to another. For example, in Tartan the type of the values used to index the elements of an array (the type of the index set) is a generic characteristic, whereas the exact bounds of the array (which values are in the index set) are attributes.

The set of attribute names associated with a type and the types of the corresponding attribute values are given in the definition of the type. For example, objects of type fixed have attributes Max, Min, Precision, and Scale.

Note that the attributes values of an object are not part of its type. It is therefore possible to write routines that operate on objects with different attributes. For example, it is straightforward to write routines that operate on arrays of arbitrary size.

It is often convenient to define families of related types with similar properties, and in which the differences can be captured through differences in generic properties. A type definition parameterized in this way can be cast as a **generic type definition**. Members of the family with distinct characteristics are distinct types.

Generic types are introduced through generic module definitions. For example,

```
generic module Blocker[T:type];
    begin
    type Block[T](Order:Int) = array(1..Order) of T;
    proc BlockIt(var B:Block[T]); begin . . . end
    end
```

defines a set of types Block[...] and a set of corresponding procedures. The definitions

```
module IntBlock is Blocker[Int];
module RealBlock is Blocker[Real];
module MyBlock is Blocker[MyType];
```

introduce, respectively, the types

```
Block[Int](Order:Int)
Block[Real](Order:Int)
Block[MyType](Order:Int)
```

each of which has an Order attribute. Note also that the procedure BlockIt is overloaded to operate on all these types, and that it is indifferent to the Order attribute of its argument.

1.5.2. Type Names

In Tartan, a <type name> may be either a simple identifier or an identifier inflected with additional type names. The <type name> so formed captures the signature of the type. For example, the <type name>s in the example above are

```
Block[Int]
Block[Real]
Block[MyType]
```

Although the definitions of these three types are closely related (they arise from instantiations of the same generic module), the types are entirely distinct.

The <type name>s for the primitive scalar and simple nonscalar types are the keywords used to declare them: **fixed, float, boolean, latch, char, set, string, actname, file.**

The <type name> for an array declared "array(a..b) of D" is "array[I,D]", where I is the <type name> of a and b. See section 1.5.3 for the derivation.

The <type name> for an enumeration declared enum[L1,L2,...Ln] is enum[L1,L2,...,Ln].

The <type name> for an activation declared activation of P is activation[P].

The <type name> for a dynamic type declared dynamic T is dynamic T.

The <type name> for a record type is based on the sequence of field names and <type name>s in its declaration. For a record declared "record[F1:T1, F2:T2, ..., Fn:Tn]" the <type name> is "record[F1:TN1, F2TN2, ..., Fn:TNn]", where the Fi are lists of field names, the Ti are types, and the TNi are type names. Bindings in the declaration do not appear in the type name. Thus, in the code

fragment

```
proc P(var x:record[a,b:Real]); begin . . . end;
var y:record[a,b:Real];
var z:record[c,d:Real];
```

variables y and z have different <type name>s and only y is acceptable as a parameter to P.

The <type name> for a variant is "variant[TT,T1->V1,T2->V2,...,Tn->Vn]", where TT is the <type name> of the tag, Ti is the i^{th} value of the tag type, and Vi is the <type name> that corresponds to the i^{th} value of the tag type. As a result, two variant <type>s are the same if they specify the same <type>s for all values of the tag. Thus for

```
type Color = enum (red, green, blue, yellow);
variant T:Color (on red -> x:Int on blue -> y:Mark(5) on others -> z:array(1..5) of Int)
```

the <type name> is "variant[Color, red->Int, green->array[Int,Int], blue->Mark, yellow->array[Int,Int]]".

The <type name> for a defined type is the type name given in the type definition, as illustrated above for Block[...].

1.5.3. Array Types

The built-in array type is in fact a generic family. Arrays have uniform properties in that every array is a structure for storing a linear homogeneous fixed-length sequence of values indexed by a given ordered set of values. However, arrays with different element types or different types of indices are distinct types.

This particular generic family of types is so common that Tartan, like most languages, provides special syntax for it. There is a set of types pre-defined as "array[IxType,EltType](r)" where IxType is the index type, EltType is the element type, and r is a (sub)range of IxType. The syntax "array(r) of EltType" is provided as an abbreviation for each such type. Thus "array(1..10) of float" means "array[int,float](1..10)". Its type name, "array[int,float]", is written "array[int] of float". Thus if we have declared

```
var V: array (1..10) of Float
var B: array (red..green) of boolean
```

the generic type of both B and V is array, but their <type name>s are different. The <type name> of B is array[int,float], whereas the <type name> of V is array[color,boolean].

The type "array(A,B) of T" is an abbreviation for "array(A) of array(B) of T". Similarly, the array accessor "V(i,j)" is an abbreviation for "V(i)(j)".

1.5.4. Declarations

The attributes of a variable become fixed at the time of its allocation. For static variables, this occurs during elaboration of the declaration. Variables of dynamic types do not themselves have attributes. The dynamically allocated objects they refer to do, however, have attributes; these are supplied whenever a constructor is executed.

The declaration of a static variable must provide both a <type name> and values for the attributes associated with that type. For example, the declaration "var V: array (m..n) of Int", which is an abbreviation for "var V: array[Int,Int](m..n)", computes the current values of m and n to obtain the range of the index set, then statically allocates a suitable block of storage. However, the program fragment

```
type Arr(n:Int) = dynamic array (1..n) of Int;
var V: Arr;
V := Arr(5)();
```

allocates the variable V with type Arr, no attributes, and all values undefined. The declaration allocates a reference to V and sets it to nil. The constructor dynamically creates a new object of type array(Int) of Int with subscript range attribute "1..5" and associates this object with variable V. A subsequent assignment to V might use a constructor with a different bound.

1.5.5. Type Checking

The type checking rule for matching actual and formal parameters is based on the types (but not the attributes) of the parameters. The actual parameter is acceptable iff the <type name> from its declaration exactly matches the <type name> of the formal parameter.

The attributes of the values returned by a function invocation are determined immediately before calling the function. They must therefore be specified in terms of input values of the function. For example, if Str is a type with attribute Length, the definition

```
func Concat(S,T: Str)R:Str; begin . . . end;
```

would not be legal, since the attributes of the functional result are not specified. The following, however, would both be legal (but would have different meanings):

```
func Concat(S,T: Str)R:Str(27); begin . . . end;
func Concat(S,T: Str)R:Str(S.Length+T.Length); begin . . . end;
```

This simplifies the implementation, but it precludes the definition of functions that return values whose attributes can only be determined during the evaluation of the function. This should not usually be a stringent constraint; in the worst case a dynamic type may be used to return the value.

1.5.6. Defining Types

A user may introduce a new type into his program with a type definition. The type definition itself merely introduces the <type name> and defines the representation of the type. Operations are introduced by writing routines whose formal parameters are of the newly-defined type. Scope boundaries, particularly module boundaries, play no role in the definition of the type. There is, as a consequence, no notion of the complete set of operations on a type.

A type definition may be parameterized with attributes. The bindings in the formal parameter list must be const or manifest. If a <binding> is omitted, it will be assumed to be const. The names of the formal parameters of the type are available throughout the elaboration of the program as constants, called attributes. They are accessed by treating the <var ident> as a record and the type attribute as a const field. Attributes for primitive types are given as part of the type definitions.

1.5.7. Operations on New Types

Operations on new types are introduced by routine definitions. These may be either routines called with normal invocation syntax or definitions for infix functions. In order to make it possible to write basic operations on the new type, Tartan provides a means of applying operations of the underlying representation to objects of the new type. Within the scope in which the type is defined, the qualifier Rep may be used to indicate that the object named by the identifier it qualifies is to be treated as if it had the underlying type. It is not exportable. This allows operations on the new type to be written using operations on its representation. When no ambiguity arises, the Rep qualification may be omitted. For example, we may write

```
type Mark = Int;
func "+"(a,b: Mark)c:Mark; begin Rep'c := Rep'a + Rep'b end;
```

Rep qualification is intended to be used within a module in order to write primitive operations and to extend operators to the new type. It is obviously possible to abuse the facility.

An assignment operator is automatically supplied for user-defined types. Although it may be invoked with any variable and value of the type, it signals the BadAssign exception if the attributes of its left and right operands are not identical or if component-by-component assignment would fail. Sizes of nonscalars are thus guaranteed to be compatible. Clearly, assignment may be well-defined in cases where this rule disallows it. Such assignment operators could be provided if user-defined assignment were compatible with the requirements.

When a module is used to encapsulate the definition of a type and its operations, the type name and some of the operations must be exported from the module. Types, named routines, field accessors for records, and variables are exported by including their names in the exports list of the module. The right to apply infix operators, constructors, subscripts, ".all", or the create command are exported by including the special names T'infix, T'constr, T'subscr, T'all, and T'create, respectively, in the exports list. Literals of enumerated types are exported automatically if the types are exported.

1.6. Parallel Processes

Parallel processes are controlled with data of two types -- activations of processes and actnames, or names of activations. An activation variable must be an instantiation of a given process; it may contain at most one activation of that process during its lifetime. An actname variable is a pointer to an activation. A single actname may be associated with different instantiations of different processes

from time to time.

Processes are similar to procedures. The syntactic distinction between procedures and processes is imposed because we believe the potential for parallel execution should be indicated explicitly in the program.

Note that activations and actnames control only the parallel control flow of the program. No synchronization is supplied with the processes; this must be coded explicitly with the primitive latches or with other, nonprimitive synchronization.

1.6.1. Activations

Activations of processes are used to control parallel or pseudo-parallel execution of instances of the named process. If P is a process and x is a variable of type activation of P, then x can contain an independently-executing instantiation of P, called an activation of P. An activation of P may be in one of several states:

- Mint: A mint activation has not yet been started up as a process. The only operations that can be performed on it are create, NameOf (i.e., the function that returns the activation's name), and the state-interrogation predicates. A newly-declared activation or actname is initialized to the literal mint.

- Suspended: A suspended activation can have no effect on any objects; in essence, it is not executing and will not execute until it is activated (see below).

- Active: An active activation is one in which it is feasible for elaboration to take place. It may affect objects, and its clock may advance.

- Dead: A dead activation admits of no further elaboration. It cannot be revived and it can play no further role in the program. An activation becomes dead when it exits normally, when it fails to handle an exception raised during its elaboration, or when it is named by a Terminate command.

The extent of an activation variable is determined by the block in which it is declared. When such a variable is declared, an activation of the named process is instantiated, set to state mint, and associated with the declared process name. The immediately enclosing block cannot be exited until all activations declared within it are dead or still mint. An activation is associated with exactly one process, but a single process may be instantiated multiple times for different activations.

If x has been declared as an activation of P and is in mint state, the statement "create x(...)" creates a new activation of P in suspended state. The formals of P are bound to the actuals supplied in the create in the same way as actuals are bound for a procedure call. If a process takes a var parameter, the corresponding actual parameter must have extent at least as great as the activation's extent. For purposes of this rule, an activation passed as a var parameter to a routine is treated as if its scope were that of the process definition. As a result, translators need no dynamic extent checking.

Except for create, all operations on activations are syntactically routine invocations. These routines control the processes and hence the parallelism by changing and interrogating the state of individual activations. They are listed in the Tartan Reference Manual.

1.6.2. Fork and Join

The extent rules require each activation to complete (exit or terminate) or still be mint before the block in which it is declared can exit. This provides an implicit join operation. A fork can be obtained with a series of creates and activates. For example,

```
begin
process P(const x:Int); begin . . . end;
var V: array(1..10) of activation of P;
for i in 1..10 do create V[i](i); activate(P[i]) od
. . .
end
```

declares ten activations of a process, uses create to start them up with different values of the input

variable (using the loop index as the input value as well as to index the array of activations), moves each activation into active state, and waits at the end of the block for the activations to terminate. After starting the activations of P, the main program may continue with other computation, monitor the progress of the activations, or simply wait for the activations to terminate.

1.6.3. Activation Names

An **actname** may name any activation. An actname variable is not permanently associated with any particular activation, and there is no requirement about the state of the activation named by an actname when the extent of that actname variable is exited or terminated. This permits routines to operate on activations without knowing what processes they are activations of. For example, it makes it possible for routines that are generally useful for managing activations to be defined in a large scope without requiring all process definitions and activation variables to include that scope. A single activation may be named by more than one actname. There is no dangling reference problem: Even though the reference (**actname**) may outlive the activation, the activation will be dead (**terminated** or **mint**) after its block is exited (and thus no unexpected computational results can be induced).[1] Since the **create** command cannot be applied to an **actname**, the process cannot be restarted.

Activation variables may not be the objects of assignments and may not appear in result parameter positions. However, each activation has a **name**, of type **actname**. This name may be obtained by invoking the function NameOf on an activation. All operations on activations except create extend to actnames. Thus, Suspend(NameOf(x)) has the same effect as Suspend(x) The special operation Me() returns the **actname** of the current process. In addition, actname variables may appear in assignments. (Thus users may write programs that operate on anonymous activations, for example to do special-purpose scheduling.) The extent of an actname variable may dominate the extent of the activation it names. If that situation arises, after the extent of the activation is exited, the actname will refer to a terminated process, and no damage can be done.

The Notify operation on activations or actnames signals the Terminate exception in the currently-executing statement of the activation named by the command. Within the activation in which it is raised, Terminate is treated like any other exception. This is the only mechanism provided by Tartan that enables one activation to interrupt another.

1.7. Unresolved Issues

We did not obtain solutions to all the Ironman requirements in the two-month period allotted to this design. In this section we sketch the way we would address the unresolved issues.

1.7.1. Machine-Dependent Code

Machine-dependent code presents two issues: definition of operations and definition of data. Tartan will permit separately-defined machine-dependent routines to be incorporated in the same way as other separate definitions. This is consistent with the Steelman requirement. We have not yet addressed the problem of machine-dependent declarations (data layout).

1.7.2. Simulation

We believe Tartan supports a programmed solution to the simulation requirement. For example, the facilities of Simula 60 can be provided for Tartan programs:

- Tartan activations can serve the same function as Simula activities.
- A coroutine call discipline may be programmed using the routines Activate and Suspend.
- A scheduler that manages simulated time can be programmed, again using operations on activations.

---

[1] The activation record itself may be allocated in the heap; it does not become eligible for garbage collection until all references have been broken. Thus no actname can become an uncontrolled pointer.

1.7.3. Definition of Integers

In the reference manual we chose fixed as a primitive and defined Int as a special case by choosing attributes appropriately. We believe it is possible to treat int as primitive and define Fixed as nonprimitive by associating range/precision bookkeeping with the operations.

1.7.4. Low-Level Input and Output

We included file as a primitive data type but did not specify its properties. Given the ability to write machine-dependent code to access the devices and the ability to use processes to maintain state (and hence to avoid, for example, re-opening a file for each operation), we believe a wide variety of low-level I/O can be implemented effectively.

1.7.5. Higher-Level Synchronization

Numerous synchronization disciplines have been proposed or are in active use. None of them clearly dominates the others; none is appropriate in all cases. We have elected to provide a very primitive synchronization tool, a latch. Conceptually, a latch is a spinlock; failure to sieze such a lock does not necessarily release the processor. By choosing a primitive mechanism, we hope to avoid pre-empting the implementation of higher-level synchronization techniques. We believe alternative mechanisms can be implemented effectively in Tartan. Indeed, we believe that this is the correct approach.

2. Programming Examples

Several sample Tartan programs are presented here. Some show the use of various features of the language; others provide programmed (nonprimitive) solutions to certain Ironman requirements.

2.1. Simple Static Data Type

A circular buffer is implemented in a vector. The definition is generic in the type of the elements; the length of the buffer is an attribute of the type. This implementation keeps a pointer to the current head of the buffer (Head) and a pointer to the element one past the current end of the buffer (Tail). All arithmetic on these pointers is done modulo the size of the buffer.

```
generic module CircularBuffers(T:type);
    begin
    exports CircBuf[T],                          ! type, attribute Size
        Clear, Append, Remove, Full, Empty,      ! routines
        BufOvfl;                                 ! exception

    type CircBuf[T](Size:Int) = record(Bf:array(0..Size-1) of T, Head,Tail: Int);

    exception BufOvfl;

    proc Clear(result C: CircBuf[T]); begin C.Head:=0; C.Tail:=0 end;

    proc Append(var C: CircBuf[T], const Val:T);
        begin
        if Full(C) then signal BufOvfl;
        C.Bf(C.Tail) := Val;
        C.Tail := mod(C.Tail+1, C.Size);
        end;

    proc Remove(var C:CircBuf[T], result Val: T);
        begin
        assert ~ Empty(C);
        Val := C.Bf(C.Head);
        C.Head := mod(C.Head+1, C.Size);
        end;

    func Full(C:CircBuf[T])F:boolean; begin F := (C.Head = mod(C.Tail+1, C.Size)) end;

    func Empty(C:CircBuf[T])E:boolean; begin E := (C.Head = C.Tail) end;

    end    ! module CircularBuffers
```

2.2. Simple Dynamic Data Type

We define a list-processing module. Each list cell contains a value of a specific type; the definition of the module is generic in this type.

```
generic module ListDef[T:type];
    begin
    exports List[T], Data, Next,          ! type and field names
        Clear, Insert, Delete, Last;      ! routines

    type List[T] = dynamic record (Data:T, Next:List[T]);

    proc Clear(result L:List[T]); begin L := nil end;

    proc Insert(var Elt:List[T], Val:T);
        begin
        if Elt = nil
            then Elt := List[T]'(Val,nil)
            else Elt.Next := List[T]'(Val,Elt.Next)
        end;

    proc Delete(var Elt:List[T]); begin assert Elt ~= nil; Elt := Elt.Next end;

    func Last(L:List[T])p:List[T];
        begin
        p := L;
        if p ~= nil then while p.Next ~= nil do p := p.Next od fi;
        end;

    end    ! module ListDef
```

2.3. Selecting Representations

Although Tartan treats types with different representations as different types, it is possible to use the variant and case facilities to define generic types that provide simiilar types with different representations. The representation is fixed during translation, when the generic definition is instantiated.

This example defines two alternative representations of queues. It has two generic parameters. The first is the type of the elements being queued, and it is used as in the previous examples. The second is a manifest constant, which is used to select which representation of queues is to be used. Since the variant is fixed during translation, there should be no loss of execution efficiency.

The two representations of queues are defined in terms of the circular buffers of section 2.1 and the lists of section 2.2.

```
generic module QueueDef[T:type, F:enum[Fix,Flex]];
    begin
    exports Queue[T],                          ! type, attribute Size
        Clear, Enq, Deq, Empty, Full,          ! routines
        QOvfl;                                 ! exception

    module Lst is ListDef[T];
    module CBf is CircularBuffers[T];

    type Queue[T](Size:Int) =
        variant manifest Fx: enum[Fix,Flex] := F
            [ on Fix -> CircBuf[T](Size) on Flex -> List[T] ]

    exception QOvfl;                       ! can only be raised on Queue[Fix]

    proc Clear(result Q: Queue[T]);
        begin
        case F on Fix -> Clear(Q(Fix)) on Flex -> Clear(Q(Flex)) esac
        end;

    proc Enq(var Q: Queue[T], const Val: T);
        begin
        case F
            on Fix  -> Append(Q(Fix), Val) { on BufOvfl -> signal QOvfl }
            on Flex -> Insert(Last(Q(Flex)), Val)
            esac
        end;

    proc Deq(var Q: Queue[T], result Val: T);
        begin
        case F
            on Fix  -> Remove(Q(Fix), Val)
            on Flex -> begin Val := Q(Flex).Data; Delete(Q(Flex)) end
            esac
        end;

    func Empty(Q:Queue[T])E:boolean;
        begin
        case F
            on Fix  -> E := Empty(Q(Fix))
            on Flex -> E := (Q(Flex) = nil)
            esac;

    func Full(Q:Queue[T])E:boolean;
        begin
        case F on Fix -> E := Full(Q(Fix).FixRep) on Flex -> E := false esac
        end;

    end    ! module QueueDef
```

2.4. Safe Data

Tartan does not provide indivisible operators for fetching and storing values. If parallel processes are operating, the programmer needs to take precautions to ensure the indivisibility of these operations. This program illustrates a solution that will work well with types for which fetching and storing the whole value makes sense.

```
begin

module Complex is assumed(ComplexLib);  ! Complex exports type Comp

generic module SafeData(T:type);
    begin
    exports Safe[T], Get, Put;          ! type name, fetch and store routines

    type Safe[T] = record (Lk:latch, Data:T);

    func Get(var S:Safe[T])R:T; begin Lock(S.Lk); R := S.Data; Unlock(S.Lk) end;

    proc Put(var S:Safe[T], var R:T); begin Lock(S.Lk); S.Data := R; Unlock(S.Lk) end;

    end;   ! module SafeData

module SafeComplex is SafeData[Comp];

var x,y,z: Safe[Comp];
Put(x, Comp'(1.,0.));
Put(y, Comp'(0.,1.));
Put(z, Get(x)+Get(y));

end;
```

Function Get takes a Safe[T] (here, a Safe[Comp]) as a var parameter. Since the Lk field is not exported from module SafeData, Get may use the procedures Lock and Unlock on that latch in order to protect the fetch.

Procedure Put specifies var parameters in both positions. Even though it does not alter R, a const specification would cause a copy.

The generic SafeData module is instantiated specifically for numbers of type Comp (the type exported by module Complex).

In the main program, the Comp constructor is used twice to generate values to store in the variables. The newly-constructed values in the calls on Put are accessible only in this program, so the constructor itself does not need to be indivisible. In the third assignment (call on Put), the addition is the addition for type Comp exported by module Complex.

3. Optional Additions to the Language

In the course of the Tartan design, we encountered a number of features that seemed attractive but could not be admitted because they violated either the Ironman requirement itself or the rule of minimality that we adopted for the design experiment. We list some of these here, indicating what they might add to the language and what they might cost.

Abbreviations for compound names. The import rule as stated can lead to the need for a substantial amount of qualification because all exported names, especially of types and routines, are potentially available pervasively. A renaming facility would reduce the need for explicit qualification. The renaming facility might involve renaming on import, or it might be a general with-clause. It would add convenience and probably improve the readability of the language. However, it would introduce a new construct in the language and introduce a new way to create aliases.

Less-than-global storage pools. As the language is defined, all dynamically allocated variables share the same heap. It would be possible to add the ability to declare a local sub-heap (zone) on the stack and allocate designated dynamic variables from it instead. There might be several zones active at once, with certain groups of variables sharing different ones. Alternatively, zones might be associated with blocks and all dynamic types defined in a block would share storage from a common zone. The cost is an additional mechanism and more complex scope rules. The benefit would be more control over dynamic variables and possibly more efficient storage recovery.

Resumable and parameterized exceptions An interrupt-style exception that has the semantics of a procedure call (resuming where it was raised) would be a useful thing to add. It would provide better control over many exception situations. Almost all the necessary mechanism must already be there to deal with the Notify command (i.e., the Terminate exception). In addition, the ability to pass parameters would be helpful, although it would complicate the syntax.

Richer control constructs. A loop exit and explicit function return could reduce the number of gotos and awkward conditional statements in programs. A richer collection of loop structures (downward counting, repeat with exitif, and so on) would add convenience. However, each such construct adds to the size of the language.

Assertions in declarations. As presently formulated, assertions are statements. It could be useful to permit them in declarations in order to check values of attributes and to guard initialization expressions. It would, however, require additional complexity in the syntax.

User-definable assignment. As noted in section 1.5.7, a default definition of assignment cannot anticipate all reasonable type definitions and all situations in which assignment makes sense. Only the programmer has the knowledge to do so. Tartan already permits infix operators to be overloaded for new types; there would be little additional cost for allowing ":=" to be overloaded as well.

References

[1] Department of Defense Requirements for High Order Computer Programming Languages, Revised "Ironman", July 1977. Appeared in SIGPlan Notices, 12, 12, December 1977 (pp. 39-54)

[2] Mary Shaw, Paul Hilfinger, Wm. A. Wulf, "TARTAN Language Design for the Ironman Requirement: Reference Manual", Carnegie-Mellon University Technical Report, June 1978.