Standardized Extensions to Modula-2

C. Pronk *
Delft University of Technology,
The Netherlands,
C.Pronk@twi.tudelft.nl

M. Schönhacker
Vienna University of Technology,
Austria,
Schoenhacker@eiunix.tuwien.ac.at

R.J. Sutcliffe
Trinity Western University,
Canada,
rsutc@twu.ca

A. Wiedemann
Gisela Gymnasium München,
Germany,
Albert_Wiedemann@compuserve.com

June 16, 1997

Abstract

This article will describe a number of extensions to the programming language Modula-2 that are in the process of being standardized by ISO. Modula-2 has been extended by facilities for generic and object oriented programming and by a set of rules to facilitate calling APIs defined in the C language. The first two extensions will be defined in part 2 and 3 of the multi-part Modula-2 standard (of which part 1 describes the base standard). The latter extension will be described in a Technical Report.

Keywords: Modula-2, Generics, Object Orientation, Interfacing to C

1 Introduction

During the standardization process of Modula-2 the group responsible for developing the ISO standard (ISO/IEC JTC1/SC22/WG13) found it necessary to extend the language with new features. These new features should make the language up-to-date and give Modula-2 similar facilities as other languages that have the same application area. The standard for Modula-2 has been set-up as a multi-part standard: i.e. the main standard [1] describes the Base Language; part 2 has been reserved for the generics extensions and part 3 will contain the object oriented extensions. These extensions have been set-up in such a way that generics and object orientation have been added to the language as pure extensions: an implementation can conform to the Base Language and to the Base Language extended with one of both, or both extensions. The standard for the Base Language has been described in Sigplan Notices in [4, 7, 10].

The most needed facilities were found to be generics and object orientation. Generic facilities provide for generic data structures such as lists and trees. In the literature many approaches to adding generics to Modula-2 have been used over the years [2, 5, 6, 9]. However, none of these approaches respected the type safeness inherent in the design of Modula-2. It was decided that facilities for generic programming should be introduced in the language in an unobtrusive way. These facilities are further described in section 2.

Object oriented extensions to Modula-2 have already been discussed in [3]. WG13 developed a set of object oriented extensions facilitating a modern programming style. The facilities added are based upon minimal changes and additions to the existing language. Obviously, such facilities cannot be added without adding additional syntax and semantics to the language. The proposed extensions

*All correspondence to be directed to the first author at POBox 356, 2600 AJ, Delft, The Netherlands

include facilities for traced objects and optional automatic garbage collection. The OO-extensions will be described in section 3.

Modula-2 programmers have always needed access to the wealth of libraries available in the C language. Although many compilers have often allowed various extensions, it was decided that a general set of rules for binding to C was desired. As such extensions require facilities that are not expressible in Modula-2 itself and are not checkable for conformance, these extensions will be described in a Technical Report. Section 4 will give a short overview of the facilities offered in this TR. Section 5 will provide a short conclusion. The concrete syntax of both proposals may be found in the appendices.

2 The Generic Extension

2.1 Introduction to the Model

Programmers often have a need for notational mechanisms that allow them to write programs in a manner that is initially free of the need to specify data types for common structures and routines. Such facilities promote code reusability and modularization and so reduce the likelihood of introducing errors when modifying old code to change the data types on which the routines operate. Languages such as Modula-3, Ada, and C++ already have generic programming facilities, and WG13 is now proposing that similar functionality be optionally added to ISO standard Modula-2.

Because the proposed new facilities are extensions, programs written to comply with the Base Language will still compile correctly except in the single instance where such old programs might employ as an identifier the one new keyword that is used to denote generic separate modules. There are some new requirements for error detection, but no new exceptions or library modules have been defined, although the proposal leaves open the possibility of adding these at a later date.

Genericity is defined at the separate module level only, by tagging both definition and implementation parts with the new keyword GENERIC. (All the contents of a generic separate module are regarded as generic, but no item can be declared as generic independently of the module containing it). Generic separate modules have formal parameters that are either types or constant values.

The process of producing a specific module with particular types or values in place of the generic ones is called refinement, and is performed by writing refining separate and/or local modules that have actual parameters compatible with the formal parameters of the generic separate module from which the refinement is being made.

Formal/actual module parameter correspondence semantics are identical to those employed in the Base Language for constant value parameters of procedures. However, there are no variable parameters allowed in generic separate module parameter lists; neither can variables be used to pass values to constant value parameters in generic separate module parameter lists. On the other hand, type names are allowed as parameters.

2.2 The New Modules

Four new kinds of compilation modules are added to the list provided in the Base Language.

2.2.1 Generic Separate Modules

Generic separate modules look like the separate modules of the Base Language, with two exceptions:
1. The reserved word DEFINITION or IMPLEMENTATION (as the case may be) is preceded by the new reserved word GENERIC.
2. The module name is (optionally) followed in the header by a formal parameter list.

The syntax and semantics of generic separate modules is in every other way identical to that of any other separate modules. As when writing a procedure, the programmer uses the formal parameter names in the expectation that they will be substituted later with actual parameters. The formal parameters of the two parts of the generic separate module are required to be the same in order to make the contract with the user explicit. (It was partly for this same reason that types and constants and not module names were allowed as parameters). Generic separate modules cannot be imported from or used as

they stand, and to attempt to do so is an error. Indeed the only action the compiler can perform on generic implementation modules is to check them for consistency; translation into code takes place after refinement and the resolution of parameters.

Some care has to be taken in the writing of the generic version of such code to ensure that the translation will eventually be possible. For instance, if a programmer employs code in the generic implementation that assumes something about a type that is passed to it via one of the formal parameters, and then a refinement is done passing an actual type inconsistent with this assumption, the error will only be detected upon attempting to translate the refinement of the implementation (i.e. after the refinement has been constructed by the refiner).

2.2.2 Refining Separate Modules

Refining separate modules are quite brief and exist only to provide the translator with the actual parameters to be used in conjunction with the generic module from which they refine. To be consistent with the Base Language and to allow for the normal development cycle (in which definition modules may exist long before their implementations) there are two compilation units; one to refine each part of the generic separate module. These must have identical (and in-scope) parameters that match the formal parameters.

If the actual parameters are located in a module, they must be qualified by that module's name. Because this is essentially a static or template mechanism, variables are not permitted as parameters.

The effect of translating a refining definition or implementation module is the same as constructing and translating a definition or implementation module (in the sense of the Base Language) obtained from the generic separate module that it refines from but with the results of evaluating the actual parameters of the refining definition module substituted for the formal parameters of the definition module of the generic separate module that is being refined.

Thus, the refined module has its imports, declarations, definitions, initialization, finalization, and exports determined by the generic separate module with the formal parameters substituted by actual parameters from the refining modules. No new such elements can be introduced in the refining module.

2.3 Module Parameters

The rules for module parameters are consistent with the Base Language rules for procedure parameters, except that there are exactly two kinds of formal module parameters: constant value parameters and type parameters. The types of the former are, as for Base Language value parameters, specified by formal types. The latter are types, and this is formally specified by the keyword TYPE which is new use of the keyword as though it were a standard identifier.

Module constant value parameters are restrictions of procedure value parameters in the same sense in the Base Language in that the corresponding actual parameters can only be compatible constant expressions. Type parameters provide a means of passing a type identifier to the refinement of a generic separate module.

2.4 Examples

Example 1: (Generic data structure)
The first example is a generic definition of a data structure. In such applications, it is expected that the structure will be manipulated independent of the kind of element it contains. All the work of programming is in the structure manipulation; the provision of the items to enter into it is a refining detail. One first provides the generic modules written in terms of the formal parameters.

```
GENERIC DEFINITION MODULE Queue (Element: TYPE; size : CARDINAL);
  PROCEDURE Enqueue (item : Element);
  PROCEDURE Serve (VAR item : Element);
  PROCEDURE Empty () : BOOLEAN;
END Queue.
```

```
GENERIC IMPLEMENTATION MODULE Queue(Element: TYPE; size : CARDINAL);
  VAR queue    : ARRAY [0..size] OF Element;
      queuePtr : CARDINAL;
  (* procedure code in terms of items of type Element*)
BEGIN (* module body initialization *)
  queuePtr := 0;
END Queue.
```

Observe the use of TYPE to denote a formal type parameter and the writing of declarations, definitions and code in terms of the formal parameters. One or more refinements can be done of each part by refining modules such as:

```
DEFINITION MODULE FacultyQueue    = Queue (Faculty.member, 200);
END FacultyQueue.

IMPLEMENTATION MODULE FacultyQueue = Queue (Faculty.member, 200);
END FacultyQueue.
```

The syntax for referring to the generic module name resembles that of a type declaration in the Base Language. The effect of processing these refinements is the same as processing copies of the generic modules with the appropriate parameter substitutions. Here, `Faculty` is the name of a module from which the data type member is, in effect, imported. The actual constant parameter could also have been obtained from a separate module.

Example 2: (An outline of a generic sort)
This example illustrates the abstraction of a generic technique applied to an existing kind of structure (an array), rather than to a user-defined structure. Here, it is the contents of the array that are generic. Again, the generic separate modules are produced first, followed by some refiner(s).

```
GENERIC DEFINITION MODULE Sort (Item : TYPE; GenCompare : CompareProc;
                                                Assign : AssignProc);
FROM Comparisons IMPORT CompareResults;
  TYPE CompareProc = PROCEDURE (Item, Item) : CompareResults;
       AssignProc  = PROCEDURE (VAR Item, Item);
  PROCEDURE Shell (VAR data : ARRAY OF Item);
END Sort.

GENERIC IMPLEMENTATION MODULE Sort (Item : TYPE; GenCompare : CompareProc;
                                                    Assign : AssignProc);
FROM Comparisons IMPORT CompareResults;
  PROCEDURE Swap (VAR a, b : Item);
     VAR temp : Item;
  BEGIN
      Assign (temp, a);
      Assign (a, b);
      Assign (b, temp);
  END Swap;

  PROCEDURE Shell (VAR data : ARRAY OF Item);
  BEGIN
  (* typical Shellsort algorithm, except that compares are done by a procedure
  Compare <actual one supplied as parameter> which returns values of type
  Comparisons.CompareResults.
  Swaps are done using the refinement of the generic swap above. *)
  END Shell;
```

```
END Sort.

DEFINITION MODULE IntSort    =
            Sort (INTEGER, IntegerInfo.Compare, IntegerInfo.Assign);
END IntSort.

IMPLEMENTATION MODULE IntSort =
            Sort (INTEGER, IntegerInfo.Compare, IntegerInfo.Assign);
END IntSort.
```

A client has:

```
FROM IntSort IMPORT Shell;
```

or, if more than one refinement to separate modules is done:

```
IMPORT IntSort, RealSort, RecSort;
```

Example 3: (A structure that is dynamic as to size, but generic as to contents).
Structures such as dynamic arrays can also be specified in a generic manner. As indicated above, however all generic elements must be refined for specific values and types prior to run-time. The only dynamic aspect is the overall size (number of items in the array).

```
GENERIC DEFINITION MODULE DynArray (Item : TYPE; nullItem: Item);
  TYPE DArray;
  PROCEDURE New (array : DArray);
  PROCEDURE Dispose (array : DArray);
  PROCEDURE MaxIndex (array : DArray) : CARDINAL;
  PROCEDURE IsNullItem (item : Item; array :  DArray) : BOOLEAN;
  PROCEDURE Assign (item : Item; array :  DArray; index : CARDINAL);
  PROCEDURE Fetch (array :  DArray; index : CARDINAL) : Item;
END DynArray.

GENERIC IMPLEMENTATION MODULE DynArray (Item : TYPE; nullItem: Item);
(* An implementation might be as a linked list of nodes that contain the addresses
  only of the data items.  The purpose of  the nullItem is initialization.  *)
END DynArray.

DEFINITION MODULE StudentArray    =
        DynArray (Students.Student, Students.nullStudent);
END StudentArray.

IMPLEMENTATION MODULE StudentArray =
        DynArray (Students.Student, Students.nullStudent);
END StudentArray.
```

2.5 Refining Local Module Declarations

The generic extensions to Modula-2 provide some additional functionality beyond the similar ones in Modula-3, namely the ability to refine as a local module. A refining local module is similar to a local module in the sense of the Base Language, and similar to a refining implementation module as shown above, but with a few changes.

The effect of translating any module containing a refining local module is the same as constructing and then translating a module in which the refining local module is replaced by a refined local module in the sense of the Base Language. The latter is constructed from the implementation module of the

generic separate module that it refines from, but with the results of evaluating the actual parameters of the refining local module substituted for the formal parameters of the generic separate module that is being refined.

Such a refining local module can be employed in any context in which the refined local module is permitted by the Base Language, including a program module, the implementation module of a separate library module (generic or not), or a local module.

Since there are no definition parts of local modules, some additional rules are necessary:
1. The imports of a refined local module are the merger of the imports of the definition and implementation parts of the generic separate module of which it is a refinement together with the actual module parameters (i.e. these values can be treated as imports).
2. Unlike a refining implementation module, a refining local module may have an export list, and once the parameter substitutions are taken into account, this becomes the export list of the refined module into its surrounding scope.

The main functionality achieved by having export lists in local refinements is to allow several refinements to be made and exported without clashes into the same scope. (Either qualified or unqualified export is permitted). However, because not all of the items named in the definition part of the generic separate module need be exported into the surrounding scope, and because the refinement is of the implementation rather than of the definition, the surrounding scope has access to the details of data types specified as opaque in the generic definition part. This permits the additional functionality of generating altered abstract data types by changing one or more procedures. For example, one could, as shown in [8] define a generic queue, then from it construct a priority queue using local module refinement of the original queue module in a new separate module (whether generic or not).

Example 4: (Multiple refinement in one scope).
Observe that the parameters may in this case be available from the surrounding scope rather than from a separate module. One of the refined routines is exported qualified; the other not.

```
MODULE Outer;
IMPORT Sort;

  MODULE CardSort = Sort (CARDINAL, CardCompare);
  EXPORT Shell;
  END CardSort;

  MODULE RealSort = Sort (REAL, RealCompare);
  EXPORT QUALIFIED Shell;
  END RealSort;

END Outer.
```

3 Object Oriented Extensions

3.1 Introduction to the Model

As Modula-2 in its original design provides for basic facilities necessary for object orientation (like data encapsulation and modularization), full object oriented facilities can be easily added to the Base Language in a very natural way. Thus the advantages of this programming paradigm are made available to the programmer in a fully upward compatible way.

3.1.1 Summary of Model Characteristics

The following list provides an overview of the OO-model.

- Access to objects: via references only.
- Arity of inheritance: single inheritance.

- Visibility modes: three modes via explicit visibility rules-hidden, family (for those developing subclasses) and public.
- Constructors/Destructors (without parameters): class initialization/finalization (similar to module initialization/finalization).
- Object allocation/deallocation: CREATE and DESTROY (for untraced classes) / implicit (for traced classes).
- Type inquiries: test for membership of a class-type family and selection in a guarded region.
- Class syntax: different from modules and records because of new concepts and similar to modules and records because of visibility rules and type nature.
- Garbage collection: on dedicated (traced) classes.
- Standard root object: none.
- Genericity: none; this is subject of a separate effort, as described in section 2 of this paper.
- Operator definition: no.
- Operator overloading: no.
- Method covariance: no.

3.1.2 Reference-based Semantic Model

The model for object-oriented programming for standard Object Oriented Modula-2 is to use reference-based semantics; i.e. object variables contain references to objects and assignment copies references and not objects themselves.

For example, if **x** and **y** are object variables, **x := y** results in **x** having the same reference to an object that **y** has; it does not copy to **x** the contents of the object referenced by **y**. Note that the consequence of this model is to allow objects to be created only dynamically.

3.2 Class Declarations

A new kind of entity is introduced, called a class. Classes are a new kind of type (called class type) – the type of reference based object – and can be used as any other type.

Classes are declared in implementation or program modules (referred to as "class declaration"). In addition to a class declaration, a class definition may occur in the corresponding definition module. The class definition specifies an external interface for a class in the same way that a procedure definition specifies an external interface for a procedure.

A class declaration or class definition opens a new scope. All identifiers declared in a class definition are also visible in a class inheriting from this class (they are referred to as family visible). The identifiers declared in a class declaration are only visible in the class declaration of a subclass in the same module; to subclasses declared in other modules they remain hidden.

An optional reveal clause allows one to specify components that may be accessed from outside the class or its descendants; they are further referred to as public components.

Furthermore, classes may be declared as abstract, and a class may optionally inherit from another class, as one would expect.

3.3 Traced and Untraced Objects

Two kinds of class are provided: traced and untraced. This allows two kinds of object to be created: traced and untraced.

A program that uses traced objects is safe from undetected dangling object reference errors, except when the tracing mechanism is deliberately subverted. The implementation guarantees this by the automatic initialization of traced object references and the automatic collection of traced objects that are no longer referenced.

A program that uses untraced objects is not safe from undetected dangling object reference errors. In such a program, it is the programmer's responsibility to prevent such an error by ensuring that the program does not use uninitialized object references or references to objects that have been destroyed.

An untraced class may have a finalization body that is executed when the object is explicitly destroyed. Traced objects are automatically destroyed in an order and at a time that is implementation dependent, and can therefore not have a finalization body.

The presence of a garbage collector and the two kinds of classes also necessitated a change at the module level: Compilation modules that contain either untraced objects or statements that threaten the ability of the garbage-collector to track references to traced objects are required to be tagged as unsafeguarded by including the keyword UNSAFEGUARDED in the module header.

If a program contains no unsafeguarded modules, it is by virtue of the garbage collection mechanism guaranteed to be safe from undetected dangling object reference errors. A program that contains unsafeguarded modules is no longer safe from such errors. In such a program, it is the programmer's responsibility to ensure that the unsafeguarded modules do not cause such errors.

An additional system module GARBAGECOLLECTION is provided as a programming interface to the garbage collector.

3.4 Examples

3.4.1 Class Definition and Declaration

This example shows the syntax of simple class definitions and declarations:

```
UNSAFEGUARDED DEFINITION MODULE Module1;
 (* class definition for "Class1": *)
 CLASS Class1;
   REVEAL                        (* visible to clients *)
     firstState, State,
     attribute2, Meth1,
     READONLY attribute1;        (* immutable by clients *)
   TYPE
     State = (starting, active, passive);
   CONST
     firstState = starting;
   VAR                           (* attributes *)
     attribute1: State;
     attribute2: REAL;
     attribute3: INTEGER;        (* accessible to subclasser only *)
   (* method definitions: *)
   PROCEDURE Meth1 (i: INTEGER);
   PROCEDURE Meth2 (r: REAL);  (* accessible to subclasser only *)
 END Class1;
END Module1.

UNSAFEGUARDED IMPLEMENTATION MODULE Module1;
 (* class declaration for "Class1": *)
 CLASS Class1;
```

```
  REVEAL
    Meth3;                (* visible in "Module1" *)
  VAR
    attribute4: INTEGER;  (* hidden entity, i.e. private to this class, but also
                             visible to subclasses in this implementation module *)
  (* method declarations: *)
  PROCEDURE Meth1 (i: INTEGER);
  BEGIN
    attribute1 := active;
    INC (attribute4, i);
    Proc1 (SELF);
  END Meth1;

  PROCEDURE Meth2 (r: REAL);
    ...
  END Meth2;

  PROCEDURE Meth3 (i: INTEGER);  (* private to this class *)
  BEGIN
    Meth1 (i);
    DEC (attribute4);
  END Meth3;

END Class1;

PROCEDURE Proc1 (c: Class1);
  ...
END Proc1;

...
END Module1.
```

3.4.2 Type Inquiries

Two methods for type inquiries have been added to the object oriented extensions: the built-in function ISMEMBER and the statement GUARD. Their use is illustrated by the following example:

```
VAR draw:  ARRAY [1..10] OF DrawObject;  (* array of objects *)
 ...

BEGIN
 IF ISMEMBER (draw, DrawPrinter) THEN (* If the dynamic type of draw *)
   DrawForEveryPage;                  (* is assignment compatible to *)
 ELSE                                 (* DrawPrinter, do DrawForEveryPage. *)
   DrawOnce;
 END;
 ...

 GUARD draw[i] AS
   screen: DrawScreen DO
     (* within this statement sequence, draw [i] is asserted to be of dynamic
        type "DrawScreen" (or a subclass).  All additional components of
        "DrawScreen" are accessible through the object denoter "screen". *)
     screen.CalcScreenSize;
```

```
    screen.DoBlink;
| printer: DrawPrinter DO
    printer.CalcPrinterSize;
| plotter: DrawPlotter DO
    plotter.CalcPlotterSize;
ELSE
  (* do whatever is appropriate *)
END;
```

4 Interfacing Modula-2 to C

Almost all Modula-2 development systems offer a way to interface other programming languages, especially C. However the solutions offered are very different, and this makes it difficult to write portable Modula-2 programs with an interface to C libraries, in a way that conforms to [1].

Modula-2 programs using interfaces derived from a set of C header files according to the rules of the proposed Technical Report will be portable between different implementations of the interface to the same C library. Because of the large number of C library interfaces (e.g. the Posix and various windowing APIs), there is considerable interest in interfacing to the C language.

The rules in the Technical Report define the transformation from C header files to Modula-2 pseudo definition modules. These pseudo Modula-2 definition modules are used to describe the interface. It is not intended to define a new binding, but to apply the standards for Language Independent Datatypes (ISO/IEC 11404), Language Independent Procedure Calling, and the Guidelines for Language Bindings (ISO/IEC 10182), to define (concrete) rules for interfacing Modula-2 to C.

In practical cases, the mapping as generated by the rules in the TR will almost never be complete. For instance, some of the preprocessor directives that tend to be present in C header files do not translate into Modula-2. However, many other things can indeed be mapped. One of the means of mapping data types is a module which maps the typical C data types to a particular Modula-2 implementation.

Note that due to the nature of the interfacing task, it turns out that not everything can be done in "regular" Modula-2. Therefore, the Technical Report also proposes a set of compiler pragmas which provide the programmer with more control of the "low-level" parts of the translation process. For instance, procedure calling conventions may have to be changed, as well as data type representations and other things.

```
DEFINITION MODULE C_Types;
 TYPE
   char = <implementation-defined CHARACTER-TYPE>;
          (* sizeof(char) = SIZE(char) *)
   signed_char = <implementation-defined INTEGER-TYPE>;
          (* sizeof(signed char) = SIZE(signed_char) *)
   unsigned_char = <implementation-defined CARDINAL-TYPE>;
          (* sizeof(unsigned char) = SIZE(unsigned_char) *)
   short = <implementation-defined INTEGER-TYPE>;
          (* sizeof(short) = SIZE(short) *)
   ...
   long_double = <implementation-defined extented REAL-TYPE>;
          (* sizeof(long double) = SIZE(long_double) *)
...

CONST
  NULL = SYSTEM.MAKEADR( <implementation-defined ADDRESS constant> );
         (* must have the C representation of NULL *)

VAR
```

```
    NULL_str [NULL] : ARRAY [0..0] OF char; (* should be treated as a constant! *)
END C_Types.
```

5 Conclusion

As can be seen from this article, a programming language standard should just be seen as a basis of work, rather than a complete, monolithic document. As a programming language is being used, users will always come up with wishes for extensions and amendments.

Extending the Modula-2 standard proved to be fairly easy, because the Base Language is extremely clearly defined and allows for orthogonal additions and amendments. The extensions for Genericity and Object Orientation were developed in parallel by different members of the working group. By virtue of this orthogonal framework they did not interfere with each other, which may be interpreted as a proof of concept for the (sometimes criticized) formal design of the Base Language Document.

Finally, we wish to thank all the individuals and organizations participating in the development of the described extensions to the Modula-2 Standard.

A The Concrete Syntax of the Generic Extension

```
generic definition module =
   "GENERIC", "DEFINITION", "MODULE", module identifier,
   [formal module parameters], semicolon,
   import lists, definitions,
   "END", module identifier, period ;

generic implementation module =
   "GENERIC", "IMPLEMENTATION", "MODULE", module identifier, [interrupt protection],
   [formal module parameters], semicolon, import lists, module block,
   module identifier, period ;

refining definition module =
  "DEFINITION", "MODULE", module identifier, equals,
   generic separate module identifier, [actual module parameters], semicolon,
   "END", module identifier, period ;

refining implementation module =
   "IMPLEMENTATION", "MODULE", module identifier, equals,
   generic separate module identifier, [actual module parameters], semicolon,
   "END", module identifier, period ;

generic separate module identifier = identifier ;

formal module parameters =
   left parenthesis, formal module parameter list, right parenthesis ;

formal module parameter list =
    formal module parameter, {semicolon, formal module parameter} ;

formal module parameter =
   constant value parameter specification | type parameter specification ;

constant value parameter specification = identifier list, colon, formal type ;
```

```
type parameter specification = identifier list, colon, "TYPE" ;

refining local module declaration =
   "MODULE", module identifier, equals, generic separate module identifier,
   [actual module parameters], semicolon, [export list],
   "END", module identifier ;

actual module parameters =
   left parenthesis, actual module parameter list, right parenthesis ;

actual parameter list = actual parameter, {comma, actual parameter} ;

actual parameter = constant expression | type parameter ;
```

B Concrete Syntax of the OO-Extensions

B.1 Class Definition

```
class definition =
   ( traced class definition | untraced class definition ) ;
untraced class definition =
   ( normal class definition | abstract class definition ) ;
traced class definition =
   "TRACED", ( normal class definition | abstract class definition ) ;

normal class definition =
   normal class header, ( normal class definition body | "FORWARD" ) ;
normal class header =
   "CLASS", class identifier, semicolon ;
normal class definition body =
   [ inherit clause ], [ reveal list ],
   normal class component definitions, "END", class identifier ;

abstract class definition =
   abstract class header,
   ( abstract class definition body | "FORWARD" ) ;
abstract class header=
   "ABSTRACT", "CLASS", class identifier, semicolon ;
abstract class definition body =
   [ inherit clause ], [ reveal list ],
   abstract class component definitions, "END", class identifier ;

class identifier = identifier ;

normal class component definitions = { normal component definition } ;
normal component definition =
   "CONST", { constant declaration, semicolon }     |
   "TYPE", { type definition, semicolon }           |
   "VAR", { class variable declaration, semicolon } |
   (normal method definition | overriding method definition),
   semicolon ;

abstract class component definitions =
```

```
   {abstract component definition } ;
abstract component definition =
   "CONST", { constant declaration, semicolon }      |
   "TYPE", { type definition, semicolon }            |
   "VAR", { class variable declaration, semicolon }  |
   (normal method definition | abstract method definition |
    overriding method definition), semicolon ;

class variable declaration = identifier list, colon, type denoter ;

normal method definition = procedure heading ;
overriding method definition = "OVERRIDE", procedure heading ;
abstract method definition = "ABSTRACT", procedure heading ;
```

B.2 Class Declaration

```
class declaration =
   ( traced class declaration | untraced class declaration ) ;
untraced class declaration =
   ( normal class declaration | abstract class declaration ) ;

normal class declaration =
   normal class header, ( normal class declaration body | "FORWARD" ) ;
normal class header =
   "CLASS", class identifier, semicolon ;
normal class declaration body =
   [ inherit clause ], [ reveal list ],
   normal class component declarations,
   [ class body ], "END", class identifier ;

abstract class declaration =
   abstract class header,
   ( abstract class declaration body | "FORWARD" ) ;
abstract class header =
   "ABSTRACT", "CLASS", class identifier, semicolon ;
abstract class declaration body =
   [ inherit clause ], [ reveal list ],
   abstract class component declarations,
   [ class body ], "END", class identifier ;

class body = module body;

normal class component declarations =
   { normal component declaration } ;
normal component declaration =
   "CONST", { constant declaration, semicolon }      |
   "TYPE", { type declaration, semicolon }           |
   "VAR", { class variable declaration, semicolon }  |
   normal method declarations , semicolon ;

abstract class component declarations =
   {abstract component declaration } ;
abstract component declaration =
   "CONST", { constant declaration, semicolon }      |
```

```
   "TYPE", { type declaration, semicolon }          |
   "VAR", { class variable declaration, semicolon } |
   abstract method declarations , semicolon ;

normal method declarations =
   normal method declaration | overriding method declaration ;
normal method declaration = procedure declaration ;
overriding method declaration = "OVERRIDE", procedure declaration ;

abstract method declarations =
   normal method declaration | abstract method definition |
   overriding method declaration ;

traced class declaration =
   ( normal traced class declaration | abstract traced class declaration ) ;

normal traced class declaration =
   normal traced class header,
   ( normal traced class declaration body | "FORWARD" ) ;
normal traced class header =
   "TRACED", "CLASS", class identifier, semicolon ;
normal traced class declaration body =
   [ inherit clause ], [ reveal list ],
   normal class component declarations,
   [ traced class body ], "END", class identifier ;

abstract traced class declaration =
   abstract traced class header,
   ( abstract traced class declaration body | "FORWARD" ) ;
abstract traced class header=
   "TRACED", "ABSTRACT", "CLASS", class identifier, semicolon ;
abstract traced class declaration body =
   [ inherit clause ], [ reveal list ],
   abstract class component declarations,
   [ traced class body ], "END", class identifier ;

traced class body = "BEGIN", block body ;
```

B.3 Reveal List

```
reveal list = "REVEAL" revealed component list, semicolon ;

revealed component list =
   revealed component, { comma, revealed component } ;
revealed component = identifier |
   "READONLY" class variable identifier ;
class variable identifier = identifier ;
```

B.4 Inherit Clause

```
inherit clause = "INHERIT", class type identifier, semicolon ;
class type identifier = type identifier ;
```

B.5 Designators

```
object selected designator =
   object variable designator, period,
   [ class identifier, period ], class variable identifier ;
object variable designator = variable designator ;

object selected value =
   object value designator, period, [ class identifier, period ], entity identifier ;
object value designator = value designator ;
entity identifier = identifier ;
```

B.6 Guard Statement

```
guard statement =
   "GUARD", guard selector, "AS", guarded list,
   ["ELSE" statement sequence],
   "END" ;

guard selector = expression ;

guarded list =
   guarded statement sequence {vertical bar, guarded statement sequence} ;
guarded statement sequence =
   [[object denoter], colon, guarded class type, "DO", statement sequence] ;
guarded class type = class type identifier ;
object denoter = identifier ;
```

References

[1] ISO/IEC 10514-1 Information Technology, Programming Languages — Modula-2, Base Language. ISO/IEC, 1996.

[2] J. Beidler and P. Jackowitz. Consistent Generics in Modula-2. *ACM Sigplan Notices*, 21(4):32–41, April 1986.

[3] J. Bergin and S. Greenfield. What Does Modula-2 Need to Fully Support Object Oriented Programming. *ACM Sigplan Notices*, 23(3):73–82, March 1988.

[4] C. Pronk and M. Schönhacker. ISO/IEC 10514-1, the standard for Modula-2: Process Aspects. *Sigplan Notices*, 31(8):74–83, Aug 1996.

[5] J. Czyzowicz and M. Iglewski. Implementing Generic Types in Modula-2. *ACM Sigplan Notices*, 20(12):26–32, Dec 1985.

[6] M. E. Goldsby. Concurrent Use of Generic Types in Modula-2. *ACM Sigplan Notices*, 21(6):28–34, June 1986.

[7] M. Schönhacker and C. Pronk. ISO/IEC 10514-1, the standard for Modula-2: Changes, Clarifications and Additions. *Sigplan Notices*, 31(8):84–95, Aug 1996.

[8] C. Pronk and R. J. Sutcliffe. Scalable Modules in Modula-2. In Hanspeter Mössenböck, editor, *Modular Programming Languages*, number 1204 in LNCS. Springer Verlag, 1997.

[9] R. S. Wiener and R. F. Sincovec. Two Approaches to Implementing Generic Data Structures in Modula-2. *ACM Sigplan Notices*, 20(6):56–64, June 1985.

[10] M. Woodman. A Taste of the Modula-2 Standard. *ACM Sigplan Notices*, 28(9), sept 1993.