Modula-3
CS 520 Final Report

Swaminathan Sankararaman and Bojan Durickovic

May 14, 2008

1 Introduction

Modula-3 is a successor of Modula-2 and Modula-2+, and influenced, among others, by Mesa, Cedar and Object Pascal. The main design goals were simplicity and safety, while preserving the power of a system-level language. The machine-level abilities combined with garbage collection make it well suited for system programming, while the explicit interfaces and safety features support development of large-scale projects.

Modula-3 has features of both functional programming and BCPL-like languages. Its key features include objects, modules and interfaces, automatic garbage collection, strong typing, generics, safety, exception handling, and multithreading. Similarly to C/C++, the language itself was kept minimal, leaving common operations such as input and output functionality to the standard library.

2 History

The story of Modula-3 begins in 1986, when Maurice Wilkes contacted Niklaus Wirth, suggesting a revision of Modula-2/Modula-2+. Wirth had already moved on to work on Oberon, but gave his consent to Wilkes for using the Modula name. The team gathered for designing the language inlcuded Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow and Greg Nelson, and the project was sponsored by Digital Equipment Corporation (DEC) and Olivetti. The language definition was completed in August 1988, and revised in January 1989.

The contribution of the Modula-3 team goes beyond the language definition itself, as the process of developing the language through committee discussions and voting was thoroughly documented [1], and represents a unique reference of that kind.

The first compiler came out of the DEC Systems Research Center soon after the language definition, and several third-party compilers were made available in the 1990s. The 1990s, however, also saw both sponsoring companies go through a series of acquisitions[1], and the language's life in the industry never quite got started. It was mostly used for research and teaching.

Modula-3 was also used for a few notable projects, such as the SPIN OS[2], and CVSup[3]. It is

[1] After most of DEC's assets were sold between 1994 and 1997 to various companies (including Oracle, Quantum Corporation, and Intel), what remained of the company was sold in 1998 to Compaq, which was acquired by Hewlett-Packard in 2002. ("Digital Equipment Corporation," *Wikipedia: The Free Encyclopedia*, accessed May 10, 2008) Olivetti, having sold away its PC business in 1997, was acquired by a Luxemburg-based company Bell S.A. in 1999, and sold to an Italian consortium two years later. In 2003, it was absorbed into the Telecom Italia group. ("Olivetti," *Wikipedia: The Free Encyclopedia*, accessed May 10, 2008)

[2] `http://www.cs.washington.edu/research/projects/spin/www/`

[3] `http://www.cvsup.org/`

interesting to note, however, that both of these projects developed their own compilers with slight modifications of Modula-3, which was partly motivated by the weakness of the available compilers.

In 2000, the company behind one of the major compilers (CM3) for Modula-3 ceased operations, and was acquired by a German company Elegosoft, which later acquired also another Modula-3 compiler, PM3.

3 Goals and Features

The motto of the Modula-3 committee when it was formed was to select simple, safe and proven features to include in Modula-3. Ultimately, Modula-3 has ended up with a mixture of features selected from other languages and a few features of its own. The features selected by the committee were geared towards the following goals, as outlined in [2]:

1. support for structuring of large programs
2. maximizing safety and robustness of programs
3. support for machine-level programming when required
4. simplicity

The above goals were geared towards a compromise between BCPL-like languages and lisp-like languages. One provided the efficiency and flexibility and the other provides a beautiful programming model while compromising on efficiency. Modula-3 attempts to eliminate the dangers of BCPL-like languages using a strong type system, while efficiency can be achieved when required by writing unsafe code which may be machine-dependent.

We will now outline the features of Modula-3 ([2], [3], [4]). Modula-3 assimilated the following features from other languages simplifying them to a large extent.

Table 1: Modula-3's ancestry

Feature	Language
Garbage Collection	lisp
Closures	lisp
Objects	simula, smalltalk
Threads	mesa, cedar
Exceptions	clu
Generics	ada
Modules	modula-2

In addition, Modula-3 possesses a few novel features which are unique to itself:

- isolation of safe and unsafe code,
- powerful and clean type system,
- layered abstractions.

Overall, objects, interfaces and modules, threads and generics are those which contribute to the goal of structuring large programs and the rest of the features are useful for safety and robustness and for system-level programming. The goal of simplicity is followed in the inclusion of all of these features. In the following subsections we focus on those features which we think are either unique to Modula-3 or are more important in contributing towards its goals.

3.1 Modules and interfaces

Modules are main building blocks of Modula-3 programs. They provide the outermost scope: in addition to variables declared within a module, only names in imported interfaces are visible.

Modules are interrelated through interfaces. An interface defines parts of a module that are publicly visible. An interface contains declarations only, implementation is always within the module that exports it.

The file structure of a program is parallel to its module/interface structure: each module is located in a .m3 file by the same name, and each interface in a .i3 file by the same name as the interface.

A typical module looks like the following:

```
MODULE m EXPORTS i1;
        IMPORT i2;
        PROCEDURE f();
        VAR x;
BEGIN
        (* module body *)
END m.
```

Here, `i1` is an interface implemented in the module `m`, while `i2` is an interface implemented elsewhere.

By convention, the main program is in a module named `Main`. Alternatively, any name can be used for the main module as long as it exports the `Main` interface.

A module can export one or more interfaces. If no interface is explicitly exported by the module, the module implicitly exports the interface by the same name as the module. However, the interface must always be explicitly defined in an .i3 file. The explicit definition of interfaces between modules is Modula-3's feature that is geared towards large programs: it helps avoid namespace cluttering.

Unless the name `n` from interface `i` is made visible directly via `FROM i IMPORT n`, names from imported interfaces are prefixed by the interface name: `i.n`.

A module may or may not have a body (startup code within the `BEGIN`/`END` block). Typically, the focus of the main module would be on the body, while that of a library module would be on the type and procedure definitions before the `BEGIN` keyword.

Modules are at the same time the smallest units that can be made generic or unsafe, and the largest units available in Modula-3: there is no larger concept such as a package.

3.2 Type system

As specified by Laverman in Chapter 3 of [5], types serve three major purposes: structuring data, specification tool and consistency checking. We will deal with each of these three with respect to Modula-3.

With regard to structuring of data, Modula-3 provides array and record types, object types and packed types. Record types are provided for structuring of different kinds of data into a single entity. Object types are provided in order to implement Object-Oriented concepts where a set of operations are defined to be performed on a set of data. This simplistic definition is expanded upon in Section 3.4. Packed types in Modula-3 allow for the specification of the number of bits used for each member of the type. These contribute to a predictable mapping onto the hardware.

With regard to the type system as a specification tool, Modula-3 provides a concept called opaque types which can both be used as a specification tool and for information hiding. These two uses are closely tied. Record and Object types specify a set of fields for a particular type.

These fields constitute what the type signifies and what can be performed with the type. For a single type, depending on an implementation which uses it, different specifications can be provided. These different specifications of a single type are implemented using opaque types in Modula-3. A programmer could specify different specifications of the same type in different interfaces which are used by different implementations. A consequence of using such a system is that parts of a specification of a type are hidden to the implementer of another specification of the same type. This constitutes the information hiding mechanism of Modula-3. We devote Section 3.3 to this topic.

With regard to the use of types for consistency checking, Modula-3 supports features such as Structural Equivalence and Strong Typing as well as keeping the type system simple and uniform. These concepts are discussed below. However, we will see that Structural Equivalence has its own share of problems and in order to resolve these, Modula-3 supports a form of name equivalence when needed.

We now discuss the features of Modula-3 which deal with consistency checking.

3.2.1 Assignability and Compatibility

Modula-3 is strongly typed. In the context of Modula-3, strong typing implies that (1) There are no ambiguous types or target typing (2) No Auto-Conversions (3) Well-defined rules for type compatibility. These are accomplished through Static Type-Checking.

```
TYPE T1 = INTEGER;
TYPE T2 = [0..100];
TYPE T3 = REAL;
VAR x: T1;
VAR y: T2;
VAR z: T3;
...
x := x*y; (* (1) No Target Typing *)
...
x := z;   (* (2) No Auto-Conversions *)
```

In the above code, Statement (1) has the expression x*z. The type of this expression depends only on the types of x and z and not on what it is assigned to. Also, the type of an expression is never ambiguous, it is always checked to see if it is so at compile-time. In Statement (2) x is assigned to z. This tries to assign an INTEGER variable to a REAL. In languages like C, this would entail an implicit conversion of the REAL to INTEGER by rounding down. This sort of implicit conversion is not supported in Modula-3. Conversions should always be explicit, i.e., known by and hence, provided by the programmer explicitly.

In general, assignability and compatibility between types is defined by a well-defined subtype relation. Note that the <: operator defines a subtype.

$$\forall \texttt{T,U} \in \text{Types} \quad \texttt{T <: U} \Longrightarrow \texttt{T} \subset \texttt{U}$$

where we consider, in the consequent, the types T and U in their *denotational view.*

3.2.2 Structural Equivalence of Types

Modula-3 uses structural equivalence instead of name equivalence. This essentially means that

$$\text{Expanded-Definition}(\mathtt{T}) = \text{Expanded-Definition}(\mathtt{U}) \Longrightarrow \mathtt{T} = \mathtt{U}$$

The expanded definitions of `T` and `U` may be infinite expansions for recursive types. The main reason for the designers choosing structural equivalence over name equivalence was to support referential transparency of types. This was accepted by a majority of the committee as being suitable for distributed programming. One possible case put forward was when there was a Remote Procedure Call with a type `T` as the type of a parameter. Now, `T` needed to be usable in the remote program, for which case, structural equivalence, i.e., referential transparency was really useful. For more details, see Chapter 8 of [2].

However, there is a drawback with structural equivalence outlined in Chapter 8 of [2] as well as in [6]. This is that the definition of a programmer may need to be unique when the implementation has to be protected from malicious clients of the module. The client could guess the structure of a type and hence, since it is structurally equivalent, access the internals. For this purpose, Modula-3 has a way of provided name equivalence through the use of the `BRANDED` keyword. This is used in the *SPIN* project for untrusted classes.

3.3 Layered Abstractions and Information Hiding

The main concept of Information Hiding is based on the principles put forth by Parnas [7] which are that

1. The specification must provide to the intended user modules, software engineering, software design all the information that he will need to use the program correctly, *and nothing more.*

2. The specification must provide to the implementer, all the information about the intended use that he needs to complete the program, and *no additional information*; in particular, no information about the structure of the calling program should be conveyed.

These principles of Information Hiding are achieved by Modula-3 through the use of *Layered Abstractions.* This term means that there are multiple extents of information provided depending on who is accessing this information. This is achieved through the use of *Revelations.* Modula-2 supported revelations in the form of opaque types but Modula-3 takes this one step further through the use of partially opaque types. Correspondingly, Modula-3 supports Partial Revelations [8].

Languages such as Oberon, C++, Java implement information hiding without completely following the principles of Parnas. These languages use access specifications (typically a finite number, e.g. private, protected and public) to specify what can be accessed and what cannot. However, there is a fundamental flaw in such a method which makes it fail to satisfy Parnas' principles. This is that the implementation details are sometimes revealed to a person reading a specification even though it is not modifiable or even accessible. One reason why this is not a good thing is because when the user of a module is aware of any of its implementation details, he might be tempted to take advantage of this fact. Subsequently, if that implementation was changed, the user would have to modify his code. Hence, the user must be provided only enough information and nothing more.

Modula-3 satisfies this requirement through the use of revelations. Consider the following code-

```
INTERFACE IntList;

TYPE IList <: IListPublic;
     IListPublic = BRANDED OBJECT
        METHODS
        create();
        insert(val: INTEGER);
END;

END IntList.
...........................................
MODULE IntList;

REVEAL IList = IListPublic BRANDED OBJECT
        list: ARRAY[1..100] OF INTEGER; count: INTEGER;
        OVERRIDES create := Create; insert := Insert;
END;

PROCEDURE Create(self: IList) = BEGIN   self.count := 0;  END Create;

PROCEDURE Insert(self:IList; val: INTEGER) =
BEGIN   self.list[self.count] := val;   self.count := self.count + 1;
END;

END IntList.
```

In this case, a person who is using the Integer List has no clue as to how it is implemented. It should not be revealed how the data is stored nor how the methods work. The interface should only specify the operations which can be performed on the List which is all it does. In the module, it is revealed that the List is, in fact, implemented using an array. Now, if the user's code is dependent on this, any modification of IntList would have to be propagated to all users. The revelation demonstrated here is a complete revelation. Modula-3 supports partial revelations wherein there would be multiple revealings of a type's details. An implementer of one revealing will not know the implementation of any revealings prior to the revealing.

This system, though elegant is not without its drawbacks. The first one, identified by [5] is that the use of types as a specification tool and their use for information hiding is combined into this revelation technique. This causes granularity problems. Another drawback, identified by Collberg [9] is that it is not possible to compute offsets of fields at compile-time. It is pointed out that SRC Modula-3 solves this by performing these computations at run-time prior to the execution of user code. This causes efficiency problems.

The main advantage of this technique is that it forces the programmer to adopt a good design which is especially useful when handling large programs which would benefit from structuring so as to enable easy management and modification of code.

3.4 Objects

Object support is a novelty in Modula-3 as compared to Modula-2. The object features are intentionally kept to a minimum: there is no multiple inheritance and no operator overloading.

Single inheritance (in conjunction with static typing) follows Modula-3's goal of a system language: this ensures that method dispatch is fast ($\mathcal{O}(1)$). Single inheritance provides an object tree with the generic object ROOT at the root. However, Modula-3 is not a pure object-oriented language: built-in types are not objects and cannot be extended by the user.

Objects are references to a data record and a method suite. As references, they can be traced and untraced (see Section 3.6), so that there are actually two disjoint object trees, one rooted at ROOT and the other at UNTRACED ROOT. Also, as references, object types can also be BRANDED in order to avoid structural equivalence to any other type of object (see Section 3.2.2).

There is no constructor (default method responsible for initializing the object) in Modula-3, by convention such methods are called `init` and are called explicitly, returning the object after initializing it: `VAR m := NEW(Matrix).init();` [10].

Method overrides are made explicit:

```
TYPE ST = OBJECT
        METHODS
                m1() := P;
                m2() := P;
        END;
TYPE T = ST OBJECT
        METHODS
                m1() := Q;
        OVERRIDES
                m2 := Q;
        END
a = NEW(T);
NARROW(a, ST).m1(); (* activates Q *)
NARROW(a, ST).m2(); (* activates P *)
```

Note that the object definition does not include method implementations: methods are bound to procedures which are defined elsewhere. Candidate procedures must explicitly use a compatible *self* (the type of which is the object type or supertype) as a first argument, while the remainder of the signature must match the method's signature. (The explicit *self* as a first argument in the method procedure has been taken over in Python.)

Modula-3's object features were used for a formal study of the theory of objects. A subset of Modula-3, named *Baby Modula-3* was designed by Martin Abadi for this purpose. [11]

3.5 Safety and Support for Unsafe Features

Since Modula-3 was designed as a systems programming language, it had to incorporate some machine-dependent features such as pointer arithmetic. In order to isolate these unsafe features from the rest of the code, two types of Modules were specified- *safe* and *unsafe*. An unsafe module is defined as one in which some error can cause a computation to crash corrupting the system. This sort of error is isolated in Modula-3 into the Unsafe Modules.

In a Modula-3 program, three types of errors can be generated [2]:

- static errors which are detected during compile-time (typically a violation of the language definition),
- checked runtime errors which are detected and reported (possibly as an exception) at runtime, and

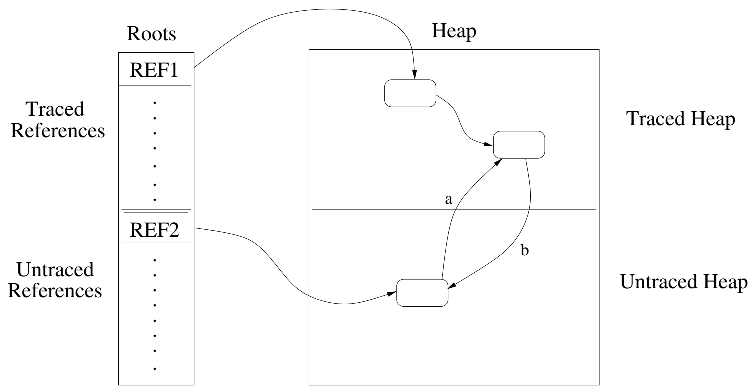

Figure 1: Traced and Untraced Heaps

- unchecked runtime errors which occur at runtime and may not be detected in which case they may cause some arbitrary operation of the program after they occur or a crash.

Modula-3 seeks to isolate unchecked runtime errors which may cause corruption in memory or some other dangerous operation from the safe code.

In order to do this are present the Unsafe Modules. These modules which are explicitly labeled UNSAFE support dangerous operations like type transfers and pointer arithmetic. See below-

```
UNSAFE MODULE M;
...
VAR X: INTEGER;
VAR Y: ADDRESS;
BEGIN
        ...
        LOOPHOLE(e,T);    (* Type Transfer of expression e to T *)
        ...
        Y = INC(ADR(X)); (* Pointer Arithmetic *)
        ...
END;
```

In the above code, INC increments address. This sort of pointer arithmetic and type transfers are allowed in Unsafe Modules. The main idea with using Unsafe Modules is to provide a clear demarcation between Safe and Dangerous code so that when a crash does occur, it will be very easy to identify which part of the program caused it. In Modula-3, this demarcation occurs when an unsafe module exports a safe interface.

In order to provide robustness, Modula-3 supports garbage collection. The garbage collection algorithm in SRC Modula-3 compiler is one we found interesting and an overview is given in the next section. Modula-3 gives a choice to the user to take advantage of garbage collection or not. This is because, since Modula-3 was designed to be a programming lanugage for large-scale systems, at the lower levels, garbage collection would be an inefficient overhead while at the higher levels, it would be extremely useful. Hence, the option is provided. For this purpose, there are two types of references: traced and untraced. Traced references point to a heap called Traced Heap which is garbage collected on by the garbage collector. Untraced references point to the untraced heap and these are not traced by the garbage collector. See Figure 1.

The other difference between Safe and Unsafe modules mentioned earlier is that in Unsafe modules, there could be a pointer from the untraced heap to the traced heap. It is the programmer's

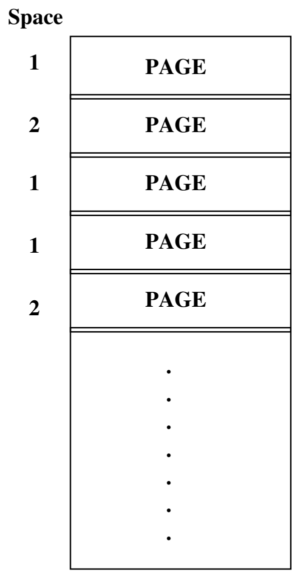

Figure 2: HEAP

responsibility to maintain the pointer and the garbage collector would not update this pointer. This corresponds to edge *a* in the Figure 1. Edge *b* is always possible since the garbage collector would move pointers also around in the traced heap.

3.6 Garbage Collector

There are two garbage collectors used in the SRC-Modula 3 compiler [4]. One is a virtual-memory based, multithreaded, incremental, conservative garbage collector, and the other is a copying collection garbage collector [12]. We now give an overview of the working of the latter and its variations from the standard copying collection algorithm.

This simple garbage collector uses an algorithm known as *Mostly Copying Collection.* The main difference between the Mostly Copying Collector and the standard copying-collector is in the finding of the roots. In a standard copying-collector, at any point of time, the garbage collector needs to maintain a set of roots (stack, register, global variables). Instead of maintaining the exact state, the garbage collector scans the state for "hints". The way these "hints" are identified forms the basis of the garbage collector's functioning.

The heap space is organized into pages of a fixed pagesize *PAGEBYTES.* Each page has an associated value *space* identifying if it is part of the *to*- or the *from*-space. In this way, the *to*- or *from*-spaces need not be contiguous but instead is a linked list of pages. See Figure 2.

Note that we are only showing the Traced Heap. Now, since the heap is organized in this manner, allocation is performed in two stages: first, it finds a free page and then it finds the free space within this page. When the garbage collector operates, it does so in two passes. In the first pass, it changes the space identifier of all hinted pages it found by scanning the state. In the second pass, it sweeps the new space for any pages which may contain items which should not be garbage collected and moves these into the new space in the same manner as the standard copying collector. Note that the garbage collector is triggered when the number of allocated pages is half the heap size.

For more details, see [12].

4 Conclusion

Modula-3 is a powerful and safe language, which balances low-level features comparable to C++ with features of high-level framework languages, such as garbage collection, safety, and explicit module/interface structure of programs.

The designers' goals were fulfilled by the very fact that every feature of the language was agreed upon by committee voting. However, it has not taken on a life in the industry, probably because of the destiny of the sponsoring companies. The actual users of the language ended up not being the intended ones: the language was designed to be used for large-scale systems, but ended up being used mostly for research and teaching.

Even though it is a very elegant language, and very appealing in theory, a major obstacle in practice is the lack of an efficient compiler. Of the several available compilers, most are open source, but due to a lack of interest, neither has achieved a high level of maturity.

Thus, it seems that Modula-3 has drifted into history too early, but its legacy remains in its strong influence on languages such as Python and C#.

References

[1] Greg Nelson, editor. *Systems Programming with Modula-3.* Series in Innovative Technology. Prentice Hall, 1991.

[2] *Systems Programming with Modula-3.* Prentice Hall Series in Innovative Technology, 1991.

[3] Geoff Wyant. Introducing Modula-3. *Linux Journal*, 1994.

[4] Farshad Nayeri. SRC Modula-3: An Elegant, Free, Open, Mature, High-Quality, Clean & Modest Programming Language, System, Library & Tool for Building Robust, Large-Scale, Multi-Threaded & Distributed Programs.

[5] Bert Laverman. *Supporting software reusability with polymorphic types.* PhD thesis, University of Groningen, 1995.

[6] W. C. Hsieh, P. Pardyak, M. E. Fiuczynski, C. Garrett, and B. N. Bershad. The interaction of access control and object-orientation in extensible systems. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8482-8.

[7] David Lorge Parnas. A Technique for Software Module Specification with Examples (Reprint). *Commun. ACM*, 26(1):75–78, 1983.

[8] Steve Freeman. Partial revelation and modula-3. *Dr. Dobb's Journal*, 20(10):36–42, October 1995.

[9] Christian S. Collberg. *Flexible Encapsulation.* PhD thesis, Lund University, December 1992.

[10] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful Modula-3 interfaces. Technical Report 113, Digital Systems Research Center, 130 Lytton Avenue Palo Alto, California 94301, December 1993. URL `ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-113.pdf`.

[11] Martin Abadi. Baby Modula-3 and a theory of objects. Technical Report 95, Digital Systems Research Center, 130 Lytton Avenue Palo Alto, California 94301, February 1993. URL `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-95.html`.

[12] Joel Bartlett. Compacting Garbage Collection with Ambiguous Roots. 1(6):3–12, 1988. ISSN 1045-3563. doi: http://doi.acm.org/10.1145/1317224.1317225.