ISO/IEC 10514-1, the standard for Modula-2: Process Aspects

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

June 27, 1996

1 Introduction

On June 1st, 1996, the long awaited standard for the programming language Modula-2 has been published as ISO/IEC 10514-1. In this article we will give some background on the process of standardization of the language and discuss some of the future activities of ISO/IEC JTC1/SC22/WG13, the working group active at standardizing Modula-2. An accompanying article [9] will give an overview of the most important clarifications, changes and additions to the language.

The language Modula-2 was originally defined in an ETH report [21]. Later definitions can be found in the report sections of the various editions of 'Programming in Modula-2' ('PIM'), the main book about the language that was written by Niklaus Wirth [22, 23, 24]. Unfortunately, these definitions differed somewhat and were incomplete and/or ambiguous in some detailed areas. An overview of problems and differences has been given in [3]. As a result from these differences compilers started to diverge, providing the impetus for standardization. One of the tasks of a standardization group is to identify these problems and try to produce a 'better' definition.

Early in the standardization process WG13 decided to use natural language (English) as well as a formal language (VDM-SL) to provide a precise definition of the language.

In section 2 we will give a short overview of the standard and show how the use of VDM-SL has helped us to give a much more precise meaning to the constructs in the language Modula-2.

Because of the use of VDM-SL, checking the standard became a necessity. Section 3 will give some details.

Section 4 will give some details about existing ISO-compliant compilers.

No doubt it will have been noticed by the reader that producing this standard has taken a long time, if not a too long time. Section 5 will explain what we learned and show some of the difficulties encountered.

Several additional extensions to the language have been proposed, and WG13 has turned some of them into 'additional work items'. These will be described briefly in section 6.

Finally, section 7 will conclude this paper.

2 A quick look at the standard

2.1 VDM-SL

VDM-SL [6] is a model oriented specification language based on denotational semantics [18, 19]. Its development started at the Vienna Laboratories of IBM around 1970 as Vienna Definition Language [8]. It was developed into META-IV by Bjørner and Jones and used for the partial definition of several programming languages [1]. The language was later extended by Jones to a more general software development method [6]. Currently VDM-SL is also subject to international standardization under ISO directives [11].

Usually, a language definition is expressed using a syntax specification notation (e.g. EBNF) and natural language to describe both the static semantics (normally checked at compile time) and the dynamic semantics (the 'meaning' of the program, computed at run time). Such natural language definitions are imprecise and ambiguous by their nature. Formal specifications can vastly improve the precision of a definition. Section 2.2 will give an example comparing a natural language specification with a formal specification written in the denotational specification language VDM-SL.

Fully introducing VDM-SL would not be appropriate for this paper. Therefore, we will only give a short description of the structure of such a specification.

The original Reports define the concrete syntax of Modula-2 in a variant of EBNF, while the Standard uses a different one [2]. As in compiler construction an abstract syntax is derived from the concrete one. This abstract syntax is used as the basis for the other parts of the formal definition. The definition consists of two main parts: the definition of the static semantics, usually called Well-Formedness clauses (*wf-predicates*) and the definition of the dynamic semantics (Meaning functions (*m-clauses*)).

In denotational semantics the meaning of a language construct is defined in terms of the meaning of its constituent parts. This process recurs to the lowest level where the meaning of a program is expressed in terms of mathematical entities like sets, maps and sequences. Section 2.2 gives the static semantics of the FOR statement (*wf-for-statement*), expressed in calls to functions checking (sub-)parts of that statement.

2.2 An example

As an example of the precision one can achieve using a formal method we present an excerpt from the definition of the FOR statement as given by Wirth [23] and the way this has been handled in the formal definition.

> The for statement indicates that a statement sequence is to be repeatedly executed while a progression of values is to be assigned to a variable. This variable is called the *control variable* of the for statement. It cannot be a component of a structured variable, it cannot be imported, nor can it be a parameter. Its value should not be changed by the statement sequence.
>
> ...
>
> The for statement
>
> ```
> FOR v := A TO B BY C DO SS END
> ```
>
> expresses execution of the statement sequence SS with v successively assuming the values A, A+C, A+2C, ..., A+nC, where A+nC is the last term not exceeding B. v is called

the control variable, A the starting value, B the limit, and C the increment. A and B must be assignment compatible with v; C must be a constant of type INTEGER or CARDINAL. If no increment is specified, it is assumed to be 1.

It can be seen that this formal definition is *(i)* incomplete with respect to the type of the loop variable and the allowed types of the initial and final expression (which are in fact called the starting value and the limit and could ambiguously be considered not to be expressions), *(ii)* does not state whether it is possible to export a loop control variable, and *(iii)* unclear about the precise meaning of 'its value should not be changed'.

The way this has been made more precise using VDM-SL will be given here without much explanation. The reader is referred to books about VDM-SL [4, 6, 11].

```
1.0  $wf\text{-}for\text{-}statement : For\text{-}statement \to Environment \to \mathbb{B}$
 .1  $wf\text{-}for\text{-}statement\ (For\text{-}statement\ (id, initial, final, step, body))\rho \triangleq$
 .2    $is\text{-}simple\text{-}identifier\ (id)\rho \land$
 .3    let $type = t\text{-}entire\text{-}designator\ (Entire\text{-}designator\ (id))\rho$ in
 .4    $is\text{-}ordinal\text{-}type\ (type)\rho \land$
 .5    $wf\text{-}expression\ (initial)\rho \land$
 .6    $wf\text{-}expression\ (final)\rho \land$
 .7    $wf\text{-}expression\ (step)\rho \land$
 .8    $is\text{-}assignment\text{-}compatible\ (type, t\text{-}expression\ (initial)\rho)\rho \land$
 .9    $is\text{-}expression\text{-}compatible\ (host\text{-}type\text{-}of\ (type)\rho, t\text{-}expression\ (final)\rho)\rho \land$
 .10   $is\text{-}constant\text{-}expression\ (step)\rho \land$
 .11   let $stype = t\text{-}expression\ (step)\rho$ in
 .12   $is\text{-}whole\text{-}number\text{-}type\ (stype)\rho \land$
 .13   let $by = evaluate\text{-}constant\text{-}expression\ (step)\rho$ in
 .14   $wf\text{-}step\text{-}range\ (by) \land$
 .15   $wf\text{-}statement\text{-}sequence\ (body)\rho \land$
 .16   $\neg\ is\text{-}threatened\text{-}in\text{-}statement\text{-}seq\ (id, body)\rho$
```

annotations Check each of the components of the for statement. The check that the control variable is declared in a variable declaration part of the block that contains the for statement is part of the consistency check for the procedure, function, or module that contains the for statement.

end annotations

The clause *Its value should not be changed ...* has been the subject of heated debates in WG13. The decision that came out of this discussion is that this test is now obligatory (note the call of the function $\neg$*is-threatened-in-statement-seq*(...) which performs this check).

The dynamic semantics (not given here) state that the value of the control variable shall be undefined after the FOR statement has been left; this even holds when the FOR statement is enclosed in a LOOP statement and the body of the FOR statement contains an EXIT statement which causes the loop to be left prematurely.

3 Testing

This section describes two issues related to verifying the contents of the standard: checking the correctness of the formal definition and specifying so-called minimal requirements clauses.

3.1 Testing the standard

As explained before, the standard contains a large amount of VDM-SL: approximately 200 type definitions, 1800 function and operation definitions and some 20,000 lines of VDM-SL code. Obviously, like writing a program, writing such a formal definition is an error-prone process. Therefore there is a strong need for mechanical checking.

Concurrent with the development of the Modula-2 standard, the formal definition of the VDM-SL language itself was also completed. To support the development of the VDM-SL, a front-end for VDM-SL was developed at Delft University of Technology. This front-end accepts a 'program' in VDM-SL and performs lexical and static semantic analysis. The input expected for such a front-end is in the so-called ASCII syntax (Interchange Syntax) of VDM-SL.

As may be seen from the example in section 2.2, a VDM-SL text may contain non-ASCII characters. The usual way to compose such a text, and the way it has been done in the standard, is by using LaTeX [7] and the VDM-SL macros [5]. The first couple of lines from the example have been reproduced below in the LaTeX input format.

```
\begin{vdm}
\begin{fn}[e]{wf-for-statement}
\signature{For-statement \To Environment \To \Bool}
\parms*{(\reccons{For-statement}{id,initial,final,step,body})\rho}
\fnapply{is-simple-identifier}{Id}\rho \And ... \\
```

As can be deduced from the examples, neither the printed format nor the input text lend themselves well as input to a tool doing a check of the VDM-SL code. The VDM-SL standard [20] contains a description of a third format expressly designed for the purpose of tool input: The Interface Syntax (formerly called ASCII syntax). Some lines of the example are given in that format below.

```
functions
wf_for_statement : For_statement -> Environment -> bool
wf_for_statement (mk_For_statement ( id,initial,final,step,body))(RHO) ==
 is_simple_identifier ( id)(RHO)  and ...
```

To derive a text in the Interface Syntax it was decided to transform the DVI output of the LaTeXsystem into ASCII by processing it with a program called `Dvi2Ascii`, an adapted version of the `dvitty` program (see figure 1). The pre and post processor in that figure were needed because of some shortcomings of the `Dvi2Ascii` tool and because of some syntactic differences between the VDM-SL as used in the Modula-2 standard and standard VDM-SL. A full account of why this set-up was chosen has been given in [12, 16, 17].

All of the VDM-SL specifications in the Modula-2 standard have been tested for syntactical accuracy and semantic constraints. Thanks to the great dedication of the project editor and the use of the LaTeX macros only a modest number of errors was found.

Unfortunately, lack of funding prohibited doing a complete type check and deriving a Modula-2 interpreter from the formal definition.

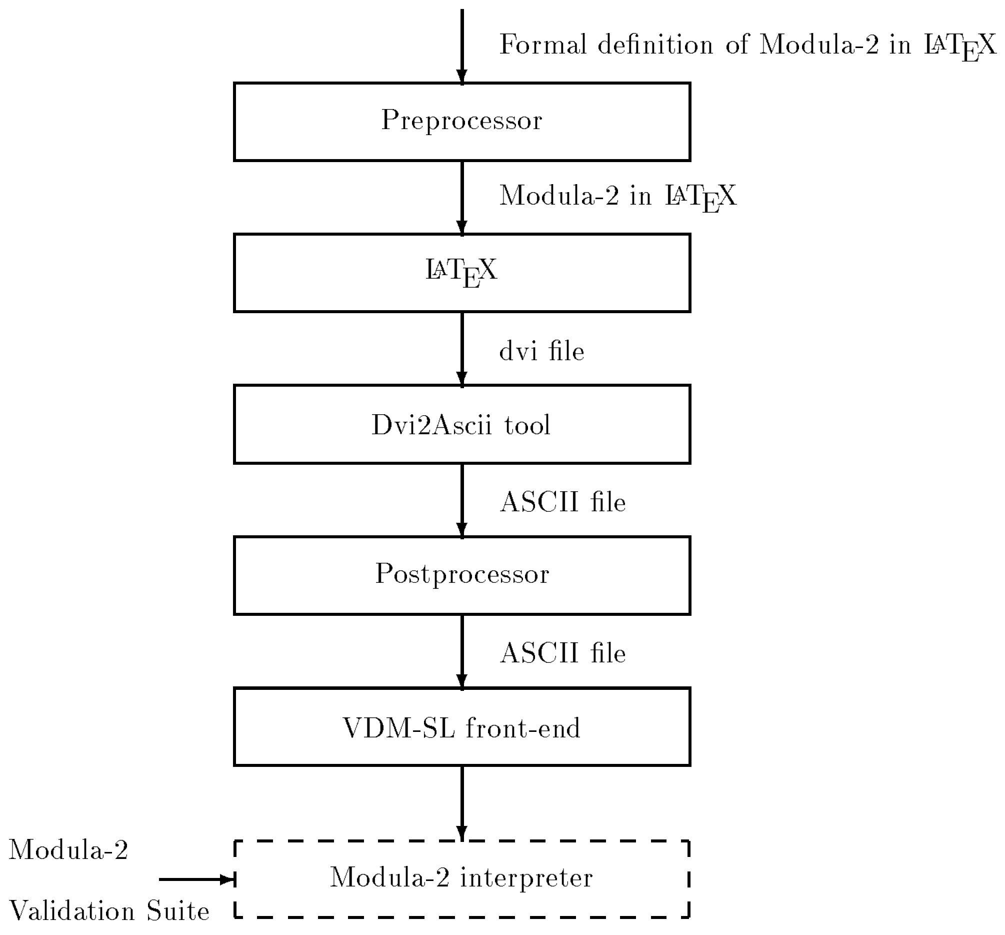

Figure 1: Tool set-up

3.2 Specifying minimal requirements clauses

A formal definition precisely specifies the syntax and semantics of a Modula-2 program. However, there are various kinds of limitations in practical compilers which can not be taken into account in the formal definition. Such limitations mostly result from constraints in buffer sizes, which in turn may result from limitations in the underlying architecture (e.g. 64K boundaries). Two examples of such limitations are: the minimal length of a string literal a compilers needs to accept, and the minimal number of nested constructs a compiler needs to accept. A set of 'minimal requirements clauses' was developed. To precisely describe such clauses a novel approach was taken. Each of the clauses was specified by a Modula-2 program that, when executed, would generate another Modula-2 program containing the specified test. The following example shows a fragment of the generating program for nested IF-THEN-ELSE-END clauses.

```
PROCEDURE GenIf (n : CARDINAL);
BEGIN
  IF n = 0 THEN
    GenStat;
  ELSE
    STextIO.WriteString ("IF b");
    SWholeIO.WriteCard (n, 1);
    STextIO.WriteString (" THEN");
    STextIO.WriteLn;
    GenIf (n-1);
    STextIO.WriteString ("ELSE");
    STextIO.WriteLn;
    GenIf (n-1);
    STextIO.WriteString ("END;");
    STextIO.WriteLn;
  END;
END GenIf;
```

In order to specify 'reasonable' values a large number of compilers was tested [14, 15]. These tests showed several unexpected results regarding the capabilities of compilers to handle the test programs.

4 ISO-compliant compilers

In this section we will give some details about ISO-compliant compilers. The authors have obtained the information in this section from various sources; they cannot be held responsible for the accuracy and completeness of the information. Mentioning a particular compiler system does not constitute a recommendation of that system by the authors of this paper. As the information in this section will be quickly outdated, the reader is referred to the FAQ `http://www.cis.ohio-state.edu/hypertext/faq/usenet/computer-lang/Modula2-faq/part1/faq.html` and `.../part2/...` for up-to-date information.

4.1 GPM

The Gardens Point Modula-2 compiler system has been developed by J. Gough at Queensland University of Technology. The GPM compiler family began as a project to make the language Modula-2 available on contemporary machines; by now Oberon-2 is also available. The compiler front-ends are based on the creation of abstract syntax trees. Native code back-ends exist for the 386/486 (SVR4, Linux, DJGPP, WindowsNT and OS/2), Sparc (Solaris), R3000 (Ultrix), DEC Alpha-AXP (OSF/1) and RS/6000 (AIX) range of systems. An MS-DOS version based upon interpretation of intermediate code is also available. The compiler and libraries are ISO compliant with the exception of the data types COMPLEX and LONGCOMPLEX.

More information may be obtained at `http://www.fit.qut.edu.au/CompSci/PLAS/GPM`.

4.2 XDS

The xTech Development System (XDS) is a programming system for Modula-2 and Oberon-2. The same programming environment is provided on all platforms ranging from MS-DOS to Windows 95/NT and Unix. An XDS translator to C and an XDS native-code compiler are available. A full set of ISO/IEC 10514-1 libraries is available.

More information is available from `http://www.iis.nsk.su/xtech/xds`.

4.3 P1

This is a full ISO/IEC 10514-1 compliant compiler producing code for the 68k-based Apple Macintosh computers. A second back-end producing C code has also been implemented. A back-end for generating code for the PowerMacs is in preparation. Except for the modules Semaphores and Processes (which may come with MacOs 8) all of the non-required library modules are available.

P1 Modula-2 has been designed as an MPW tool; to generate PowerPC native application the MPW C/C++ compiler as well as the Metrowerks compilers are supported.

The full set of MacOs APIs is included. In non-standard mode, object oriented extensions are available.

More information is available from the address contained in the FAQ.

4.4 Stony Brook

This system is a Windows-95/NT hosted integrated development system (compiler, linker, librarian, make, debugger etc.). At the time of writing of this article the system is in beta test stage. The commercial release is expected to be available during the summer of 1996. The implementation will be a full implementation of the standard with various extensions (which can be switched off). A complete set of library modules as described in the standard will be provided. Native code can be generated for DOS, DOS32, OS/2, WIN16 and WIN32.

Further information is available from the address contained in the FAQ.

5 Standardization experiences

The ISO standardization process requires a future standard to go through a number of stages (New Work Item NWI, Committee Draft CD, Draft International Standard DIS and, finally, International Standard IS). A transition to the next stage can be made after a successful international vote has taken place. The whole process takes four years, at a minimum. The process is based upon technical experts meeting each other in person, trying to reach consensus on the technicalities of the future standard.[1] Unfortunately, it is ill-defined what consensus means. While these technical experts are nominated by National Bodies (e.g. NNI, ON, DIN, ANSI/IEEE), their technical opinion may not be the same as their National Bodies' opinion.

In the ideal situation members of a working group are users of the language, people having a commercial interest in the language (like compiler builders) and academics using the language for teaching or academic research. In the usual case however, the users are missing.

Unfortunately, there are no obvious criteria for optimality of language constructs and so one finds several contradicting interests like ease of use/ease of learning, commonality with constructs in other languages (ease of switching), safety, orthogonal language design and time to market, all leading to long and heated discussions.

In the long duration of the standardization process sometimes people lose interest, even get disappointed, get ill or change jobs. WG13 members have been through some of these things, none of which contributed to a quick realization of the standard.

[1]Meanwhile, more effective standardization procedures are possible; also email correspondence is much more common now.

6 Further work

WG13 works on a number of topics aimed at extending the language and/or the libraries. In this section we will give a short introduction to each of them. A general precondition to language extensions is that they should remain as close as possible to the original goals and style of the language. Additionally, no existing programs which adhere to the base standard should be invalidated. WG13 has taken utmost care to comply with these requirements. Please note that the items described here are under development and have not yet been agreed upon in any official ISO voting process (except for the creation of the projects), so they may be subject to change or even deletion.

WG13 maintains a WWW home page located at `http://sc22wg13.twi.tudelft.nl/`. This home page gives access to current documents and the FAQ. Current documents are also available via anonymous ftp from `dutiba.twi.tudelft.nl` in the directory `/pub/wg13/`. Please note that this is all copyrighted material. Furthermore, ISO rules restrict the accessibility of any material as soon as it has reached the DIS (Draft International Standard) status.

6.1 Object orientation

Modula-2 was intended to be a Systems Programming Language, which meant an emphasis on efficiency and access to machine facilities. Nevertheless, the use of Modula-2 for general programming and teaching necessitated an extension with features for object oriented programming.

The OO model chosen adds classes as a new syntactic construct. All objects are accessible via references only. The inheritance mechanism chosen is single inheritance, multiple roots. Additionally, syntax and semantics for safe/unsafe modules and traced/untraced variables as well as garbage collection have been developed (à la Modula-3 [10]).

6.2 Generic facilities

The lack of a facility for generic programming has so far hampered full exploitation of Modula-2. The model chosen for introducing generics into Modula-2 is based upon so-called Generic Definition Modules and Generic Implementation Modules which are refined into regular definition and implementation modules using refining definition and implementation modules. Local modules may also be refined. Note that the term 'refinement' was chosen for generics because 'instantiation' was already in use by the OO extensions. Types and constants may be used as generic parameters, which results in facilities similar to those provided by C++.

6.3 Bindings

One of the factors determining the success of a language is the number of application platforms it can access, or in more modern terminology the number of APIs (X11, CORBA, POSIX, Windows, ...) that is accessible. First, a set of modules providing access to the POSIX API [13] was developed, but unfortunately the WG member in charge of this project had to withdraw from these activities.

It was quickly realized that providing interfaces to every available API would be impossible. On the other hand it was noted that currently most of the API specifications are provided in the C language. Therefore it was considered more efficient to develop a set of guidelines for bindings to APIs defined in the C language. These guidelines will probably be published as a technical report.

6.4 Pragmas

Currently, the standardized syntax allows pragmas (compiler directives) to be inserted in program texts, but does not give any semantics to them. Portability of programs would benefit from a standard set of pragmas. A working paper is currently under consideration in WG13.

6.5 Other work items

Two other work items ('Further support for concurrent programming' and 'Support for commercial programming in Modula-2') have little support at the moment, and at its recent meeting, the working group decided to ask for their termination.

7 Conclusion

Given that the example in section 2.2 pertains to just a very small part of the language, and considering the various problems described in section 5, it will now hopefully be clearer to the reader why the development of the standard took so long. However, this understanding should not turn into resignation about possible inadequacies of the standardization process. The authors still feel that programming language standards are necessary, and that the result for Modula-2 was worth the effort.

Interestingly, problems tend to show in places where one would not have expected them at first, e.g. in trying to define the term 'consensus' before being able to reach one. Having said that, this is probably one of the points that can not be addressed by guidelines. Instead, the success or failure of a project in international standardization (as in other areas) largely depends on individuals and their efforts to try to understand other people's views, as well as their willingness to put work into proposals that may in the end not find consensus and be dropped again.

The authors gratefully acknowledge the contributions of many colleagues, in particular the former WG13 convenors Roger Henry and Mark Woodman, the convenor of the VDM-SL working group and author of large parts of the VDM-SL in the Modula-2 standard Derek Andrews, the editor of the first draft Don Ward, and everyone who has endured in the long and sometimes tedious process of standardizing Modula-2.

References

[1] D. Bjørner and C.B. Jones. *Formal Specification and Software Development.* Prentice Hall, 1982.

[2] BS 6154:1981, Method of defining Syntactic Metalanguage, 1981.

[3] B. J. Cornelius. Problems with the Language Modula-2. *Software - Practice & Experience*, 18(6):529–543, 1988.

[4] J. Dawes. *The VDM-SL Reference Guide.* Pitman, 1992.

[5] I. P. Dickinson. Typesetting VDM-SL with $\mathrm{V_{DM}S_L}$. NPL document obtainable via ftp.

[6] C.B. Jones. *Systematic Software Development using VDM, Second Edition.* Prentice Hall, New York, 1990.

[7] L. Lamport. $\LaTeX$*, a Document Preparation System.* Addison-Wesley, 1994.

[8] P. Lucas. On the Formal Description of PL/I. *Ann.Rev.Aut.Progr.*, 6, 1969.

[9] M. Schönhacker and C. Pronk. ISO/IEC 10514-1, the standard for Modula-2: Changes, Clarifications and Additions. *Sigplan Notices, this issue*, 1996.

[10] G. Nelson. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[11] N. Plat and P.G. Larsen. An Overview of the ISO/VDM-SL Standard. *ACM SIGPLAN Notices*, August 1992.

[12] N. Plat, C. Pronk, and M. Verhoef. The Delft VDM-SL Front End. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal software development methods; 4th International Symposium of VDM-Europe*, number 551 in LNCS. Springer Verlag, 1991.

[13] ISO/IEC 9945-1:1990, POSIX System Interface. ISO/IEC, 1990.

[14] C. Pronk. Stress Testing of Compilers for Modula-2. *Software - Practice & Experience*, 22(10):885–897, 1992.

[15] C. Pronk. Specifying Minimal Requirements Clauses for Programming Languages Standards using VDM-SL. *Computer Standards and Interfaces*, 15(4):325–336, 1993.

[16] C. Pronk, N. Plat, and A. W. W. M. Biegstraaten. Checking the formal definition of Modula-2. In E. Hill, editor, *Safety Through Quality Conference*, 1994.

[17] C. Pronk, N. Plat, and A. W. W. M. Biegstraaten. The Use and Construction of Tools for Checking Large Language Definitions. *High Integrity Systems*, 1996. To be published.

[18] D.A. Schmidt. *Denotational Semantics, A Methodology for Language Development.* Allyn and Bacon Co., 1986.

[19] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Mass, 1977.

[20] VDM-Specification Language, Base-Language. ISO/IEC DIS 13817-1.

[21] N. Wirth. Modula-2. Technical Report 36, ETH Zürich, 1980.

[22] N. Wirth. *Programming in Modula-2, second corrected edition.* Springer-Verlag, Berlin, 1982.

[23] N. Wirth. *Programming in Modula-2, third corrected edition.* Springer-Verlag, Berlin, 1985.

[24] N. Wirth. *Programming in Modula-2, fourth corrected edition.* Springer-Verlag, Berlin, 1988.