Standards for Non-Executable Specification Languages

P. G. Larsen\textsuperscript{1} and N. Plat\textsuperscript{2}

\textsuperscript{1}The Institute of Applied Computer Science (IFAD), Forskerparken 10, DK-5230 Odense M, Denmark
\textsuperscript{2}Delft University of Technology, Faculty of Technical Mathematics and Informatics, P.O. Box 356, NL-2600 AI Delft, The Netherlands

This paper discusses the impact of the standardisation of (non-executable) specification languages; standardisation can increase the interest in, and acceptance of, a specification language, and it stimulates the development of tool support for such a language. It is argued that a specification language should preferably be formally defined. The ISO/VDM-SL standard (under construction) is used as an illustration. The fact that many specification languages are non-executable causes problems in the areas of conformance and compliance. These problems are touched upon.

Received June 1992

1. INTRODUCTION

The use of formal languages for system specification is increasing, and this has led to the development of a number of standards for such languages. This raises the question of how standards for specification languages can best be defined. In this paper we shall discuss the standardisation of (in particular non-executable) specification languages. We shall argue in favour of formally defining such standards, and illustrate our arguments using the ISO/VDM-SL standard (under construction). We shall also analyse some problems related to conformance and compliance.

1.1 Formal versus informal specification

Specifications are used to express the requirements of a product. When the development of a product starts it is customary to begin by writing a specification of what the product should do. Currently, such specifications are typically formulated in natural language. The main advantage of natural language is that it is easy to read and write; the main disadvantage is that ambiguity is possible because of the very richness of natural language. Thus there is a danger that specifications formulated using natural language may be interpreted in a different way than was intended by the writer of that specification. An unambiguous notation should be used to avoid this situation.

The best-known examples of unambiguous notations are programming languages, where compilers provide an unambiguous interpretation of the syntax and semantics of the language. However, when different compilers give different semantics to the same notation, the problem is not solved, and therefore standards have been defined for a large number of programming languages. In this way it is possible for commercial tool builders to develop compilers with a syntax and semantics corresponding to the users’ expectations. The main disadvantage is that by expressing a specification using a (standardised) programming language the level of abstraction often is too low; too much emphasis is put on algorithmic detail, and this distracts attention from essential points in the requirements expressed in the specification. Although some modern programming languages provide limited data abstraction facilities, this kind of abstraction is often sacrificed in order to gain efficient access to the data (e.g. using pointers).

Another kind of unambiguous notation with a high level of abstraction is provided by specification languages. These languages are notations dedicated to describing specifications. Some specification languages have a mathematical foundation, and are therefore called formal specification languages. Algorithm abstraction is obtained in specification languages by means of implicit definitions which are impossible to use in programming languages because (in general) they are not executable. In this way we can specify what should be done, instead of specifying how to do it. An important property of such a specification language is the notion of loose specification. Loose specifications denote a choice among a range of legal results. In many cases several different results can be equally valid.

In general there is a danger in limiting the power of the notation used for specifications to the point where all their constructs can be executed, and where it is impossible to express looseness.* Programming languages do not give the proper amount of abstraction, while specification languages do.

1.2 Using specification languages in industry

Some of the non-executable specification languages supported by well-known formal methods (e.g. VDM\textsuperscript{1, 7, 13} and Z\textsuperscript{20}) have been used as ‘paper and pencil’ tools. The strength of using specification languages in this way is that it is easy to extend the notation of the specification language to suit a particular paradigm or application area.\dagger There is a trade-off here, because computer-based tools supporting such a specification language require a standard notation. A standard for a specification language makes the development of mechanised support more attractive for tool vendors, and if such specification languages are to be used in large-scale industrial applications, it is necessary to have computer-based tools supporting these languages.\dagger Although there are many prototype tools from research projects, there is still a lack of high-quality commercial tools. Fortunately

* There are so many different paradigms and application areas that it is not possible to make a coherent specification language which includes them all.

\dagger Clearly tools are not the only necessary requirement for applying formal methods. The users must also be educated in using the specification language. However, the whole aspect of technology transfer of formal methods goes beyond the subject of this paper.
the situation is improving, and we shall return to this in Section 3.

1.3 Using specification languages in standards

A standard for a product specifies which requirements the product must satisfy to comply with the standard. Currently most standards are defined using a natural language. The informal nature of such descriptions makes misunderstandings possible, and also makes it impossible to verify compliance with that standard.

In the case of programming languages most standards only formally specify the syntax by means of BNF-like descriptions. The semantics of the various constructs in the programming language is still normally explained informally in a natural language. In order to check that a compiler complies with the standard, a test suite is made which the compiler must deal with appropriately. However, such a test suite cannot prove that a compiler complies with a standard, it can only make it very plausible. Furthermore, if a compiler does not comply with user expectations in a specific case, an informal specification of the language does not necessarily clarify the problem. A formal specification, however, will always give an unambiguous answer to the question.

Although usually presented in contrast to each other, these two kinds of description — formal and informal — are best viewed as complementary: either a formal description can be explained by natural language annotations, or a natural language description can be supplemented by formal descriptions. If a standard uses a formal description for specifying its contents (simply accompanying the description with natural language comments) all advantages of the formal description also hold for the standard as a whole. Alternatively, if a standard specifies its contents in a natural language and uses a formal description of some of its parts, it is only for these parts that properties can be proved, not for the entire standard.

We believe that the introduction of formal descriptions in standards will start with the latter approach, giving some advantages. However, the major aim for future standards must be the former approach, which enables formal verification of compliance with a standard. This is in line with the conclusion from a BCS working group on formal methods in standards.

In the telecommunications industry it is nowadays recognised that it is an advantage to have standards for the specification languages that are used. Within ISO (the International Standards Organisation) two standards have been developed for such specification languages (LOTOS11 and ESTELLE12). CCITT (the Comité Consultatif International Téléphonique et Télégraphique) have also standardised the SDL specification language, which is used to describe distributed systems.

2. EXAMPLE: THE VDM-SL STANDARD

As an example of how a standard for a specification language can be defined we will present a short overview of the VDM-SL standard in this section. Two main VDM books were used as baseline documents for the standard. At the start of the standardisation process it had been decided that the standard itself should as far as possible be formally defined, in order to make it as precise as possible. However, even though the aim and the basis of the standard were clear, it took a long time to produce the standard. This was mainly due to the number of different existing VDM dialects, and to the lack of knowledge about the semantics of the combination of all the different constructs; in this way the standardisation work itself clarified many points about the VDM specification language.

A formal language \( L \) can be defined by a tuple:

\[
L = (S_L, \mathcal{D}_L),
\]

where \( S_L \) denotes the set of all valid sentences \( s \) in the language (usually implicitly defined by a combination of a context-free grammar and a function \( \mathcal{D}_L \), removing those sentences from the language generated by the grammar which are not well formed), and \( \mathcal{D}_L \) is a function:

\[
\mathcal{D}_L : s \mapsto M_{\text{set}}
\]

which provides a dynamic semantics (meaning) to a sentence of that language. Because non-executable specification languages have abstraction features such as looseness, the meaning of a sentence (specification) can be given as a set of models \( M \).

The standard for VDM-SL follows this scheme, but has some additional components as well. The complete definition of the standard for VDM-SL can be divided into a number of major components: a syntax (at three levels of abstraction), symbol representations, a static semantics, a dynamic semantics and a syntax mapping. The relation between these components is shown in Fig. 1.

The latest version of the standard can be found in Ref. 6. Below we shall present a short overview of the standard and the connections included in Fig. 1 (for a more thorough overview see Ref. 16).

2.1 The syntax

The main component of the 'user interface' of VDM-SL is formed by its syntax, which exists in two forms.

(1) EBNF rewriting rules. The EBNF formalism is nowadays the most accepted form for defining the concrete syntax of a formal language, in particular because of the possibilities for automatically generating parsers from such a definition.

(2) VDM-SL type definitions. This form is called the Outer Abstract Syntax (OAS). The OAS was introduced because it can be used by parts of the standard which are defined in terms of VDM-SL functions. This is clearly not

---

* An outstanding exception to this is the Modula-2 standard, which is currently defining the semantics formally using VDM-SL.

† An example is the Office Document Architecture standard, where a formal specification is given as an addendum to the standard.

‡ Currently the specification language supporting the Vienna Development Method (VDM) is being standardised under the auspices of both ISO and BSI.
possible for the EBNF form, because EBNF is an entirely different formalism. The OAS is used both by the static semantics \( \mathcal{D} \) (subsection 2.6) and by the syntax mapping functions \( \mathcal{S} \) (subsection 2.4).

The syntax is the starting point which is used for the definition of all other relevant aspects of VDM-SL in the standard.

2.2 Symbol representations

A VDM-SL specification has a defined representation on paper, computer screens, etc. The required representation of the symbols used in a specification is defined in the standard. Currently, two different representations exist as follows.

(1) A mathematical representation. The mathematical representation provides elegant symbols, clearly distinguishing between keywords and reserved words, and resembling generally accepted mathematical notation as much as possible.

(2) An ASCII representation. An ASCII representation has been defined to make automatic processing of VDM-SL specifications possible.

The relation between the syntax and the symbol representations is defined by a lexis \( \mathcal{L} \).

2.3 The core abstract syntax

In addition to the outer abstract syntax, the standard also contains another abstract syntax representation for VDM specifications which is less complicated than the OAS: the Core Abstract Syntax (CAS). The CAS is used for the definition of the formal semantics of the language (the dynamic semantics \( \mathcal{D} \), subsection 2.5), and it was introduced because the formulae for the formal semantics are less complicated and thus easier to understand when defined over the CAS than when defined over the OAS.

2.4 The syntax mapping

There is a large gap between the concrete representation of a VDM-SL specification (OAS) and the representation over which the meaning of a specification has been defined (CAS). Therefore, the standard contains a formal definition of a syntax mapping \( \mathcal{S} \) from a specification in terms of the OAS to a specification in terms of the CAS. A preliminary definition of \( \mathcal{S} \) can be found in Ref. 17.

2.5 The dynamic semantics

All VDM specifications that can be represented in the core abstract syntax are given a formal meaning by the dynamic semantics \( \mathcal{D} \). The definition of \( \mathcal{D} \) is based on set theory, and the mathematical notation which is used has been fixed. A number of operators are used to build a domain universe containing all valid 'values' which can be expressed in VDM-SL. On top of this domain universe a collection of semantic domains is defined. These semantic domains are used in the actual definition of \( \mathcal{D} \).

The meaning of a VDM specification can be regarded as a set of models, because specifications can be loose. Loose specification is a technique offered by VDM-SL allowing the user to specify highly abstract components which can be implemented with different functionality. The implementation must simply have a functionality that can be said to implement the loosely specified construct according to certain implementation relations for VDM-SL.

The actual definition of \( \mathcal{D} \) is given in a denotational way, without using the traditional style of explicitly constructing the denotation. Instead, first the set of all possible models is created, and then, by examining the syntactic specification, this set is restricted to those models that can be considered the denotation of the specification. This technique offers a relational style of denotational semantics.

\( \mathcal{D} \) is a total function over all specifications which can be represented in the CAS.
2.6 The static semantics

VDM specifications that are syntactically correct according to the EBNF rules do not necessarily obey the typing and scope rules of the language. The standard, therefore, provides a formal definition of the well-formedness of a VDM specification: the static semantics of the language. The static semantics has itself been formulated in VDM-SL.

The VDM specifications that have at least one model in the dynamic semantics can be considered as those which are well formed. In general, it is not statically decidable whether a given VDM specification is well formed or not. The static semantics for VDM-SL differs from the static semantics of other languages in the sense that it only rejects specifications which are definitely not well formed, and only accepts specifications which are definitely well formed. Thus the static semantics for VDM-SL attaches a well-formedness grade to a VDM specification. Such a well-formedness grade indicates whether a specification is definitely well formed, definitely not well formed, or maybe well formed.

3. TOOL SUPPORT FOR SPECIFICATION LANGUAGES

The development of tool support for formal languages has evolved to a large extent in parallel to the development of those languages, and tools have become increasingly more important. Compilers, for example, remove the tedious and error-prone task of transforming a program expressed in a programming language into a program expressed in machine code. In fact, it is inconceivable that the current production of software could be achieved without such tools. For formal specification languages a wide range of tools can be envisaged. These tools can be divided into three categories as follows.

• Syntactic support. Syntactic tools are tools that can be used for the manipulation of formal specification(s) (fragments). Examples of such tools are structure editors, cross-reference generators and pretty-printers.

• Semantic support. Semantic tools can either be used to used to manipulate the semantics of specifications or to validate the correctness of the specifications. Such tools can, for example, be used to develop new specifications from existing ones, or to 'execute' specifications. Semantic tools typically serve as active vehicles during the development of those languages, and tools have become increasingly more important. Compilers, for example, remove the tedious and error-prone task of transforming a program expressed in a programming language into a program expressed in machine code. In fact, it is inconceivable that the current production of software could be achieved without such tools. For formal specification languages a wide range of tools can be envisaged. These tools can be divided into three categories as follows.

• Pragmatic support. Pragmatic tools are used to support the management of the development of a specification. Typical examples of such tools are tools for version control, configuration control, journaling, status reporting, etc.

Having a standard for a specification language has a large impact on the availability of tools. Again, looking at VDM-SL, quite soon after the standardisation had started tools became available that supported the draft standard. At the VDM '91 conference a wide range of tools were demonstrated supporting the standard. Apparently, the introduction of a standard stimulates the development of tools to a large extent. A number of reasons can be given as follows.

• A formally defined standard leaves no space as to the interpretation of the language. It is therefore relatively straightforward to efficiently implement tools supporting the language.

• The availability of a standard can be seen as a recognition by a wide community that the language in question is 'mature' and accepted. This is an indication of the market potential of the language and tools supporting the manipulation of that language.

• The standard defines the requirements that specifications expressed in that language must meet, which makes them interchangeable between different tools.

The claim that a tool 'supports' a standard, however, raises another question: what does it mean for a tool to comply with a standard? The simple answer is: a tool complies with the standard if the syntax and the semantics manipulated by that tool comply with that standard. Unfortunately, some tools which intuitively would be included following this definition do not comply. Consider, for instance, a prototyping tool for VDM-SL. Since VDM-SL in general is a non-executable language, such a tool can only support a subset of the language, i.e. an executable subset. Furthermore, the logic employed by standard VDM-SL is a three-valued logic which cannot be executed using a sequential programming language. Thus a conditional and/or logic must be employed which satisfies the standard semantics in all cases where the definedness of a logical expression can be found by evaluating the subcomponents from left to right.* A prototyping tool for VDM-SL necessarily accepts a language which semantically differs slightly from the standard semantics. What is needed is a mechanism for relating these language and tool variations to the standard they claim to support.

4. CONFORMANCE AND COMPLIANCE

Tools for the manipulation of a specification language can differ in the type of support they provide, and in the exact definition of the language that they claim to support. As such, both the type of support provided by a tool, and the language supported by the tool, can be individually related to the standard. For example, a type checker need only be related to the syntax and static semantics described in the standard, whereas for the language supported by the tool it can be claimed that it is the same language as described in the standard, and thus also has the same dynamic semantics. It is therefore meaningful to make a distinction between language conformance to – and tool compliance with – a standardised specification language SL (Fig. 2).

In the following subsections we shall discuss these individually.

4.1 Language conformance

Language conformance is especially important when the

* As an example of a case where the conditional and/or logic does not satisfy the standard semantics we can mention a disjunction between something undefined (⊥) and true. According to the standard semantics, an expression like ⊥ v true denotes true, whereas an interpreter using the conditional and/or logic will enter an infinite recursion and thus yield ⊥.
specification language is merely used as a paper-and-pencil tool. A language $L$ is said to 'conform' to a standard if it has the same syntax and semantics as the language defined in the standard ($SL$); in other words: that language is the language defined in the standard. As we saw in the previous section, sometimes it can be useful to consider a language that is to a large degree similar to the one defined in the standard, but has some (syntactic or semantic) deviations or extensions to the standard language.* Although such a language does not conform in the strict sense to the standard language, it is possible to classify deviations and extensions.

We consider language conformance at three levels as follows.

Full conformance. A language $L$ with the same syntax, static semantics and dynamic semantics as defined by the standard, fully conforms to the standard language $SL$. More formally:

$$SL = S_{SL}$$

Extended full conformance. A language $L$, of which a subset can be defined having the same syntax, static semantics and dynamic semantics as defined by the standard, can be regarded as an extension of the standard language $SL$. In order to claim extended full conformance to the standard language, the extensions must be formally related to the standard language. More formally:

$$\forall s \in S_{SL}. \ Retrieve_{L \rightarrow SL}(s) = Retrieve_{SL}(s)$$

where the function $Retrieve_{L \rightarrow SL}$:

$$Retrieve_{L \rightarrow SL} : M_{L} \rightarrow M_{SL}$$

is a function defining the relationship between models from the domain universe of $L$ and models from the domain universe of $SL$.

Partial conformance. A language $L$ that is similar to the language described by the standard, but in some respect is a (syntactic or semantic) deviation from the standard language $SL$, can claim partial conformance to the standard language, provided that the deviations have been formally defined. In the extreme, the consequence of this definition would be that almost any language would 'partially conform' to the standard language, and therefore additional restrictions are required, limiting the kind of deviations that are allowed. For example, it could be required that deviations are only allowed if the

* We do not take deviations arising from machine dependencies into account, e.g. limitations on the length of identifiers, nesting depth, etc.

4.2 Tool compliance

Tools can provide different kinds of support for a specification language, and therefore tool compliance can be defined at different levels as follows.

Syntactic compliance. A tool can be syntactic-compliant to the standard if the type of support provided by the tool can be formally related to the syntax of the standard language. More formally, a tool $T$ is said to be syntactic-compliant with a standard language $SL$ if it can be shown that:

$$T = T_{SL}$$

Static-semantic compliance. A tool can be static-semantic-compliant to the standard if the type of support provided by the tool can be formally related to the static semantics of the standard language. More formally, a tool $T$ is said to be static-semantic-compliant with a standard language $SL$ if it can be shown that:

$$\forall s \in S_{SL}. \ Retrieve_{L \rightarrow SL}(s) = Retrieve_{SL}(s)$$

Dynamic-semantic compliance (or full compliance). A tool can be dynamic-semantic-compliant to the standard if the type of support provided by the tool can be formally related to the dynamic semantics of the standard language. More formally, a tool $T$ is said to be dynamic-semantic-compliant with a standard language $SL$ if it can be shown that:

$$\forall s \in S_{SL}. \ Retrieve_{L \rightarrow SL}(s) = Retrieve_{SL}(s)$$

This classification also provides an indication of the usefulness of a tool. For example, a parser for the language (which provides only a limited form of support, i.e. syntactic support) can only claim syntactic compliance to the standard, never full compliance.
A prototyping tool for an executable subset of a (generally) non-executable specification language could for example claim partial conformance to the standard of the language supported by the tool (because the dynamic semantics of such a language is necessarily different from the dynamic semantics of the standard non-executable specification language), whereas the tool itself could claim both syntactic and static-semantic compliance to the standard; full compliance cannot be reached.

4.3 Checking compliance

Having defined what it actually means for a tool to comply with a standard, the question of how claimed compliance can be checked becomes important. A number of ways can be envisaged to check the compliance of a tool with the standard as follows.

The use of test suites. The use of test suites for checking the compliance of tools (most notably compilers) for programming languages is well known; it is the traditional way of checking compliance. The strategy consists of providing a significant number of tests, which the tool must process in the way described by the standard. Using the same strategy for checking the compliance of tools for non-executable specification languages is not without difficulties, because the expected behaviour of a tool supporting non-executable aspects of the language cannot be checked in the same way, as illustrated by the earlier-mentioned example of a prototyping tool. The test suite strategy is, therefore, useful for checking the syntactic and static-semantic compliance, but it can only be used for an executable subset for dynamic-semantic compliance of tools.

Proving compliance. Carrying out formal proofs showing that a tool complies with the standard is possible in theory, provided the specification language has been formally defined. Unfortunately, since such formal definitions can be very complex and large (e.g. the semantics of VDM-SL comprises roughly 200 pages of formulae alone!), it will in most cases not be worth the investment, even when proof assistants are available. We do not foresee that it will be economically feasible to carry out such proofs in the near future.

Falsification. The basic idea behind falsification is that a tool complies with the standard unless proved otherwise. Such an approach is not so unreasonable as it may seem at first sight, because tools designed with no serious intent to comply with the standard would soon be falsified, i.e. specifications would be constructed which are not correctly processed by that tool. The major disadvantage of this method is that the burden of 'checking' compliance does not lie with the tool vendor, but with the standardisation body.

Rigorous arguments. This is perhaps the most pragmatic way of checking compliance. If a tool vendor can show that there is a systematic translation from the (formal) definition of the standard to the tool, it is reasonable to assume that such a tool complies with the standard (the tool may still contain bugs, but that is a different matter). So, for example, when a parser for a specification language is based on a parser generating system, syntactic compliance can be checked by comparing the underlying grammar to the grammar defined in the standard. Of course, compliance cannot be ensured following this strategy, but it can be made plausible.

Although as yet there is no easy way to ensure or check compliance, if a specification language has a formal definition it is potentially easier to check compliance than when no formal definition is given. Clearly it is impossible to prove compliance with a standard without a formal definition.

5. CONCLUDING REMARKS

The point we have made in this paper is that it is worthwhile to standardise specification languages which in general are non-executable. We believe that such standards should be formally defined, as has been done for the VDM-SL standard. We have also shown that for such non-executable specification languages it is not obvious how conformance to and compliance with a standard should be defined. This is mainly due to the way such languages are used and the different kind of tools which can be produced supporting them. Therefore it makes sense to use a notion of extended and partial conformance for languages which have a large degree of similarity to the standard specification language. In the same way it makes sense to have different levels of tool compliance because the nature of the tools can be very different.

At the same time we also have to admit that it is our experience that it takes considerable effort to define a standard for a specification language in a fully formal manner. However, we hope that experience of the VDM-SL standardisation can be used for other standards as well. One of the major achievements of the standardisation of VDM-SL is that a number of unclear points about VDM-SL have been clarified. On the tools side we think that it is interesting to see how a standard for VDM-SL has stimulated the development of new tools using the standard language. We think that these are important benefits resulting from the standardisation of specification languages.

Acknowledgements

We should like to thank Michael Andersen, René Elmstrom, Kees Prønk and Marcel Verhoef for their valuable suggestions to improve this paper.


---

**Book Reviews**


There are two sorts of compiler courses: the one presents and compares alternative techniques for each phase of compilation; the other takes a single technique from each phase and deals with it in detail. The comparative course aims to give the student the knowledge needed to choose the correct set of techniques for a given language and application, and of necessity will involve small examples and exercises to illustrate each point. The single-model course is geared more towards the student acquiring the skills and experience needed to implement a complete compiler, special cases and all. One could say that the first course gives the science, and the second the engineering, and that both are necessary and complementary. With compiler writing-taking being edged out of the curriculum these days, it is unlikely that there will be room for two. The choice of which course to go for is therefore important.

Paul Cockshott’s book is firmly in the second camp. It presents an integrated set of techniques for compiling languages in the Algol/Pascal tradition into abstract machine code, which is then assembled on a PC, the whole system using Turbo Pascal and a special Toolbox written by the author. One can scarcely get more specific, and so one wonders what the value is of sharing these tools and techniques outside the university department where they were developed.

The value is that the book treats compiler writing seriously, addressing the issues which make or break a production compiler, but which are usually ignored in textbook examples. Specifically, the book considers object-oriented heads, raster graphics, the integrated editing environment, persistent data, classes, dynamic linking and modifying memory allocation, in addition to the usual data types, control structures and procedures of a block-structured language.

The vehicle for discussion is the language Persistent S-algol, which is introduced in Chapter 2. As its name suggests, PS-Algol supports dynamic data declaration, implicit storage management, recursion and powerful input–output facilities. As such, it provides a greater challenge for the compiler writer than Pascal.

Chapter 3 looks at overall compiler strategy, and describes how the compiler will be built up from a one-pass PS-Algol to abstract machine translator followed by an assembler, and stresses how the extra step aids portability. Chapter 4 gives a brief (but modern) overview of the theory behind grammars, leading into Chapter 5, which deals with lexical analysis via regular grammars and a finite state machine. Here we encounter for the first time the Compiler Writer’s Toolbox, with different programs and units being brought into play for efficient state-driven lexical analysis. The lexemes produced as a result are then classified and stored in a symbol table as discussed in Chapter 6.

Syntax analysis begins in Chapter 7, and the method employed is recursive descent based on BNF. Analysis of the basic control structures and expressions is covered, before Chapter 8 returns to the symbol table. The Toolbox once again provides the framework for the more sophisticated types, class hierarchies and graphics in PS-Algol. The author discusses how some of these features (for example, first-class procedures) had to be eliminated from current versions of the language in order that the compiler could run on a small PC. On the other hand, colour screens were deemed sufficiently ubiquitous to cause the raster graphics to be retained.

Chapter 9 introduces the S-Algol abstract machine and goes through all the steps necessary to generate code for S-Algol’s rich data areas: stack, volatile heap and persistent heap. Chapter 10 follows this up with a discussion of the assembler, which is in itself a very useful stand-alone section. Chapter 11 goes into more detail on heap management in a dynamic declaration environment (unlike Pascal’s), and Chapter 12 discusses the integrated editor, which provides for interactive compiling and good error reporting.

Finally, Chapter 13 looks at linking a program for execution and loading it into memory, once again using the Toolbox to good effect. Some of the issues discussed here are very real – interfacing to the DOS exec unit, use of COMMAND.COM and getting back exec error codes – and are usually simplified out of compiler courses. Their inclusion adds greatly to the value of the book.

The comprehensive set of appendices rounds off the book, and includes details of how to acquire the software from the author. There are several minor errors in the diagrams and text, but overall the book is well written and presented. Only the early chapters have exercises, and there are no answers. Some of the exercises relate to using the Toolbox to write a Pascal compiler. Since S-Algol, which is the example in the book, is in a sense a super-set of Pascal, it would probably be better for students to tackle a more advanced language, for example Ada or C++.

I would certainly recommend this book for compiler engineering courses. The strangeness of S-Algol is adequately compensated by the opportunity it gives to examine more difficult compilation issues. In addition, the Toolbox together with the book could be a useful adjunct to a software practitioner’s library.

**J. Bishop**

Pretoria, South Africa

---


