Standards for Non-Executable Specification Languages

Peter Gorm Larsen
The Institute of Applied Computer Science (IFAD)
Forskerparken 10, DK-5230 Odense M, Denmark

Nico Plat
Delft University of Technology
Faculty of Technical Mathematics and Informatics
P.O.Box 356, NL-2600 AJ Delft, The Netherlands

May 27, 1992

Abstract

This paper discusses the impact of the standardization of (non-executable) specification languages; standardization 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 why 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.

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 how standards for specification languages can best be defined. In this paper we will discuss the standardization of (in particular non-executable) specification languages. We will argue in favor of formally defining such standards, and we will illustrate our arguments using the ISO/VDM-SL standard (under construction). We will also analyze 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 by 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 (standardized) 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 are the *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 of their constructs can be executed, and where it is impossible to express looseness (see [Hayes&89]). 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 [Jones90], [Dawes91], [Andrews&91], and Z [Hayes87], [Spivey92]) 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[1]. 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 mechanized 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[2]. Although there are many prototype tools from research projects, there is still a lack of high-quality commercial tools. Fortunately, the situation is improving and we will 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

[1] There exists so many different paradigms and application areas that it is not be possible to make a coherent specification language which includes them all.

[2] 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 and we will not go further into this subject here.

languages is still normally informally explained in a natural language[3]. 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 kind of descriptions – 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 proven, not for the entire standard[4].

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 [Ruggles90].

In the telecommunication industry it is nowadays recognized 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 (LOTOS [ISO8807], and ESTELLE [ISO9074]). CCITT (the Comité Consultatif International Téléphonique et Télégraphique) have also standardized the SDL [SDL] specification language, which is used to describe distributed systems.

Summarizing, we believe that if (standardized) specification languages are used in standards, more accurate and useful standards can be achieved. If such formal descriptions are accompanied by comments in a natural language the standard will also be understood by people who lack knowledge of the formal specification language that has been used.

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[5] in this section. Two main VDM books ([Bjørner&78] and [Bjørner&82]) were used as baseline documents for the standard. At the start of the standardization process it had been decided that the standard itself should as much 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 standardization work itself clarified many points about the VDM specification language.

A formal language L can be defined by a tuple:

[3] An outstanding exception to this is the Modula-2 standard [BSI/Modula2] which currently is defining the semantics formally using VDM-SL.

[4] An example is the Office Document Architecture standard [ISO8613], where a formal specification is given as an addendum to the standard.

[5] Currently the specification language supporting the Vienna Development Method (VDM) is being standardized under the auspices of both ISO and BSI (see [BSIVDM91]).

$L = (S_L, \mathcal{DS}_L)$

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 $\mathcal{SS}_L$ removing those sentences from the language generated by the grammar which are not well-formed), and $\mathcal{DS}_L$ is a function:

$\mathcal{DS}_L : s \to M\text{-set}$

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 Figure 1.

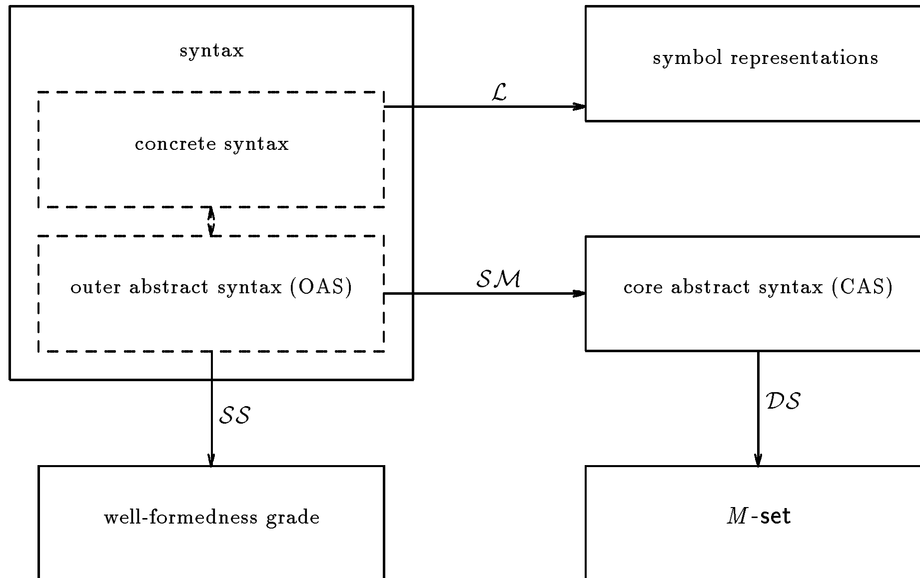

Figure 1: Structure of the VDM-SL standard

The latest version of the standard can be found in [BSIVDM91]. Below we will present a short overview of the standard and the connections included in the figure above (for a more thorough overview see [Plat&92a]).

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 *O*uter *A*bstract *S*yntax *(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 possible for the EBNF form, because EBNF is an entirely different formalism. The OAS is used both by the static semantics $\mathcal{SS}$ (subsection 2.6) and by the syntax mapping functions $\mathcal{SM}$ (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:

1. A *mathematical representation.* The mathematical representation provides elegant symbols, clearly distinguishing between keywords, 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 *C*ore *A*bstract *S*yntax *(CAS).* The CAS is used for the definition of the formal semantics of the language (the dynamic semantics $\mathcal{DS}$, 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 $\mathcal{SM}$ from a specification in terms of the OAS to a specification in terms of the CAS. A preliminary definition of $\mathcal{SM}$ can be found in [Plat&92b].

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* $\mathcal{DS}$. The definition of $\mathcal{DS}$ 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 $\mathcal{DS}$[6].

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

[6]Notice that $\mathcal{DS}$ is a total function over all specifications which can be represented in the CAS.

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 $\mathcal{DS}$ [Larsen92] 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.

2.6 The static semantics

VDM specifications that are syntactically correct according to the EBNF rules do not necessarily obey the typing and scoping rules of the language. The standard, therefore, provides a formal definition of the well-formedness of a VDM specification: the *static semantics* $\mathcal{SS}$ [Bruun&92] of the language. The static semantics has itself been formulated in VDM-SL.

The VDM specifications having 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:

- *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 for specifications or proofs, type checkers, cross-reference generators, and pretty-printers.

- *Semantic support.* Semantic tools are tools that can either be used to manipulate the semantics of specifications or to validate the correctness of the specifications. Such tools can e.g. be used to develop new specifications from existing ones, or to 'execute' specifications. Semantic tools typically serve as active vehicles during development, or for verification of a development step. Examples of semantic tools are theorem provers, compilers, prototyping tools, semantic analysis tools and transformational tools.

- *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 standardization had started, tools became available that supported the draft standard [Plat&89]. At the VDM'91 conference [VDM91] 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:

- 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 e.g. 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[7]. 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 standardized specification language *SL* (Figure 2).

In the following subsections we will discuss these individually.

[7] 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 $\bot \vee \mathsf{true}$ denotes **true** whereas an interpreter using the conditional and/or logic will enter an infinite recursion and thus yield $\bot$.

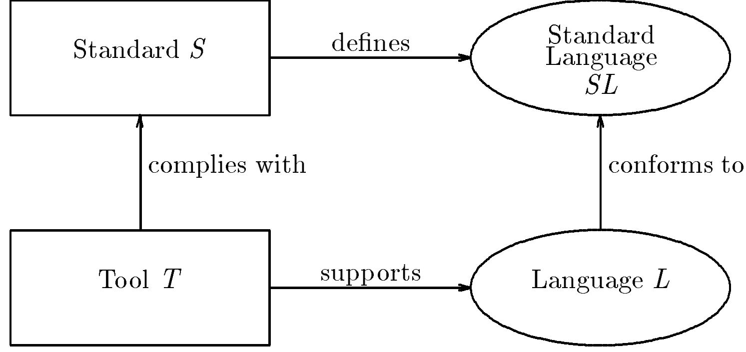

Figure 2: Conformance and compliance

4.1 Language conformance

Language conformance is especially important when the 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[8]. 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:

- *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:

  $$S_L = S_{SL} \land \forall\, s \in S_L \cdot \mathcal{DS}_L(s) = \mathcal{DS}_{SL}(s)$$

- *Extended full conformance.* A languages 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:

  $$\forall\, s \in S_{SL} \cdot \mathit{Retrieve}_{L \to SL}(\mathcal{DS}_L(s)) = \mathcal{DS}_{SL}(s)$$

  where the function $\mathit{Retrieve}_{L \to SL}$:

  $$\mathit{Retrieve}_{L \to SL} : M_L\text{-}\mathsf{set} \to M_{SL}\text{-}\mathsf{set}$$

  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

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

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 semantics of the standard language indicates non-termination:

$$S_L = S_{SL} \land \forall\, s \in S_{SL} \cdot Retrieve_{L \to SL}(\mathcal{DS}_L(s)) \neq \mathcal{DS}_{SL}(s) \;\Rightarrow\; \mathcal{DS}_{SL}(s) = \bot$$

This definition ensures that the meaning of a sentence in L differs from the meaning of that same sentence in SL, because SL has no 'appropriate' meaning for that sentence.

These three types of conformance make it possible to relate languages in a sensible way to the language defined in the standard.

4.2 Tool compliance

Tools can provide different kinds of support for a specification language, and therefore *tool compliance* can be defined with a hierarchy at different levels:

- *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:

  $$S_T = S_{SL}$$

- *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:

  $$\mathcal{SS}_T = \mathcal{SS}_{SL}$$

- *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:

  $$\mathcal{DS}_T = \mathcal{DS}_{SL}$$

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:

- *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 that 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 proven otherwise. Such an approach is not so unreasonable as it may seem at first sight, because tools designed with no serious intend 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 standardization 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, then 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, e.g. 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 then it is potentially easier to check compliance than when no formal definition is given. Clearly it is impossible to prove compliance to a standard without a formal definition.

5 Concluding Remarks

The point we have made in this paper is that it is worthwhile to standardize 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 are 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 from the VDM-SL standardization can be used for other standards as well. One of the major achievements of the standardization 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 standardization of specification languages.

Acknowledgments

We would like to thank Michael Andersen, René Elmstrøm, Kees Pronk, and Marcel Verhoef for their valuable suggestions to improve this paper.

References

[Andrews&91] Derek Andrews and Darrel Ince. *Practical Formal Methods with VDM.* McGraw Hill, September 1991.

[Bjørner&78] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language.* Volume 61 of *LNCS*, Springer-Verlag, 1978.

[Bjørner&82] Dines Bjørner and Cliff B. Jones. *Formal Specification & Software Development. Series in Computer Science*, Prentice-Hall International, 1982.

[Bruun&92] Hans Bruun, Bo Stig Hansen and Flemming Damm. *The Static Semantics of VDM-SL.* Technical Report, ID/DtH, 1992.

[BSI/Modula2] *JCT1/SC22/WG13 Draft Proposal DP 10514. Third Working Draft Modula-2 Standard.*

[BSIVDM91] *VDM Specification Language – Proto-Standard.* Technical Report, British Standards Institution, March 1991. BSI IST/5/50.

[Dawes91] John Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[Hayes87] Ian Hayes, editor. *Specification Case Studies.* Prentice-Hall International, 1987.

[Hayes&89] I.J. Hayes, C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 330–338, November 1989.

[ISO8613] *Information Processing – Text and Office Systems – Office Document Architecture (ODA) and Interchange Format.* Volume parts 1-6, ISO, 1988. Draft International Standard ISO/DIS 8613/1-6.

[ISO8807] *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* Technical Report, International Standards Organisation, 1989. ISO8807.

[ISO9074] *Information Processing Systems — Open Systems Interconnection — Estelle — A Formal Description Technique Based on an Extended State Transition Model.* Technical Report, International Standards Organisation, August 1987. ISO9074.

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM (second edition).* Prentice-Hall International, 1990.

[Larsen92] Peter Gorm Larsen. *The Dynamic Semantics of the BSI/VDM Specification Language.* Technical Report, The Institute of Applied Computer Science, February 1992.

[Plat&89] Nico Plat and Hans Toetenel. *Tool support for VDM.* Technical Report 89-81, Delft University of Technology, November 1989.

[Plat&92a] Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. *Sigplan Notices*, 27(8), August 1992.

[Plat&92b] Nico Plat and Hans Toetenel. *A formal transformation from the BSI/VDM-SL concrete syntax to the core abstract syntax.* Technical Report 92-07, Delft University, March 1992.

[Ruggles90] C.L.N. Ruggles, editor. *Formal Methods in Standards: A Report from the BCS Working Group.* Springer-Verlag, 1990.

[SDL] *Recommendation Z.100. CCITT Specification and Description Language SDL.* CCITT, 1988.

[Spivey92] Mike Spivey. *The Z Notation – A Reference Manual (Second Edition).* Prentice-Hall International, 1992.

[VDM91] S.Prehn and W.J.Toetenel, editors. *VDM'91: Formal Software Development Methods. Lecture Notes in Computer Science*, Springer Verlag, October 1991. Two volumes: LNCS 551 (Symposium proceedings) and LNCS 552 (Tutorials).