Development of an Industrial Strength Grammar for VDM

Tiago Alves and Joost Visser

{tiago.alves,joost.visser}@di.uminho.pt

Technical Report DI-PURe-05.04.29

2005, April

PURe
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-05.04.29
Development of an Industrial Strength Grammar for VDM by Tiago Alves and Joost Visser

Abstract

This report describes the development of an industrial strength grammar for the VDM specification language. We present both the development process and its result. The employed methodology can be described as iterative grammar engineering and includes the application of techniques such as grammar metrication, unit testing, and test coverage analysis. The result is a VDM grammar of industrial strength, in the sense that it is well-tested, it can be used for fast parsing of high volumes of VDM specifications, and it allows automatic generation of support for syntax tree representation, traversal, and interchange.

1 Introduction

Formal specifications are software artifacts. Some specifications are small, scribbled on paper, and thrown away when coding starts. However, when used as integral part of software development, specifications, like programs, go through a long-term evolution process involving initial development and prolonged maintenance. As a result, specifications are vulnerable to legacy problems, just as programs are.

In this report, we approach the VDM specification language (VDM-SL) from a software legacy perspective. This is justified because:

- VDM-SL specifications have been developed in industrial context and are relied upon by companies to ensure the quality of their software systems.
- Tool support for VDM-SL developed in the past is no longer available or actively maintained (Adellard's SpecBox, which is still available, was originally released in 1988 and the latest version is from 1996; IFAD's VDMTools are no longer available and the lastest release was in 2000).

As is the case with legacy programming languages, additional tool support can alleviate such legacy problems. In particular, the specification analogues of program understanding, reverse engineering, and migration could benefit from appropriate tool support.

For the development of such tool support, we advocate a *grammar-centered* approach. In such an approach, the grammar of the language in question is used for much more than single-platform parser generation. The grammar is used to generate parsers for different platforms (i.e. programming languages), but also pretty-printers, abstract and concrete syntax tree representations, serialization and interchange components, and syntax tree traversal support.

Unlike traditional Yacc-style parser specifications, grammars that are used in such a central role need to be semantics-free, and they must be neutral with respect to target platform. Also, since any grammar change potentially leads to changes in many tools that depend on it, these grammars must reach a high level of maturity before tool development starts. We contend that for the development of grammars with such characteristics the use of advanced grammar engineering techniques such as grammar metrics, grammar unit testing, and coverage analysis are essential.

In this report, we describe the development of a high-quality grammar of the VDM-SL language. In Section 2 we describe the tool-based

grammar engineering methodology that we employed. In Section 3 we describe the actual development process and its intermediate and final deliverables. In Section 4 the metric values for the VDM grammar are compared with the values for a series of other grammars. Finally, the report is concluded in Section 5. The full SDF grammar of the ISO VDM-SL language is included in an appendix.

2 Grammar Engineering

Grammar engineering is an emerging field of software engineering that aims to apply solid software engineering techniques to grammars, just as they are applied to other software artifacts. Such techniques include version control, static analysis, and testing. In this section we discuss the grammar engineering techniques that we adopted, and how we adapted them to the specific process of developing an SDF grammar for VDM-SL. In Section 5.2, a more general discussion of related grammar engineering work is provided.

2.1 Grammar-centered tool development

In traditional approaches to language tool development, the grammar of the language is encoded in a parser specification. Commonly used parser generators include Yacc, Antlr, and JavaCC. The parser specifications consumed by such tools are not general context-free grammars. Rather, they are grammars within a proper subset of the class of context-free grammars, such as LL(1), or LALR. Entangled into the syntax definitions are semantic actions in a particular target programming language, such as C, C^{++} or Java. As a consequence, the grammar can serve only a single purpose: generate a parser in a single programming language, with a singly type of associated semantic functionality (e.g. compilation, tree building, metrics computation). For a more in-depth discussion of the disadvantages of traditional approaches to language tool development see [5].

For the development of language tool support, we advocate a *grammar-centered* approach [12]. In such an approach, the grammar of a given language takes a central role in the development of a wide variety of tools or tool components for that language. For instance, the grammar may be used to generate parsing components to be used in combination with several different programming languages. In addition, the grammar serves as basis for the generation of support for representation of abstract syntax, serialization and de-serialization in various formats, customizable

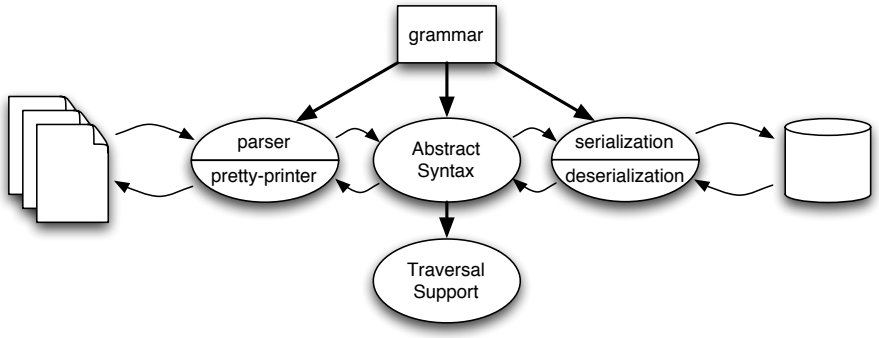

Fig. 1. Grammar-centered approach to language tool development.

pretty-printers, and support for syntax tree traversal. This approach is illustrated by the diagram in Figure 1.

For the description of grammars that play such central roles, it is essential to employ a grammar description language that meets certain criteria. It must be neutral with respect to target implementation language, it must not impose restrictions on the set of context-free languages that can be described, and it should allow specification not of semantics, but of syntax only. Possible candidates are BNF or EBNF, or our grammar description of choice: SDF [9, 22].

The syntax definition formalism SDF allows description of both lexical and context-free syntax. It adds even more regular expression-style construct to BNF than EBNF does, such as separated lists. It offers a flexible modularization mechanism that allows modules to be mutually dependent, and distribution of alternatives of the same non-terminal across multiple modules. Various kinds of tool support are available for SDF, such as a well-formedness checker, a GLR parser generator, generators of abstract syntax support for various programming languages, among which Java, Haskell, and Stratego, and customizable pretty-printer generators [3, 23, 18, 14, 11].

2.2 Grammar evolution

Grammars for sizeable languages are not created instantaneously, but through a prolonged, resource consuming process. After an initial version of a grammar has been created, it goes through an evolutionary process, where piece-meal modifications are made at each step. After delivery of the grammar, evolution may continue in the form of corrective and adaptive maintenance.

A basic instrument in making such evolutionary processes tractable is version control. We have chosen the Concurrent Versions System (CVS) as the tool to support such version control [7].

In grammar evolution, different kinds of transformation steps occur:

Recovery: An initial version of the grammar may be retrieved by reverse engineering an existing parser into a grammar description format, or by converting a language reference manual, available as Word or PDF document, or perhaps only in hardcopy, into such format.

Error correction: Making the grammar complete, fully connected, and correct by supplying missing production rules, or adapting existing ones.

Extension or restriction: Adding rules to cover the constructs of an extended language, or removing rules to limit the grammar to some core language.

Refactoring: changing the shape of the grammar, without changing its semantics, i.e. without changing the language that is generated by it. Such shape changes may be motivated by different reasons. For instance, changing the shape may make the description more concise, easier to understand, or it may enable subsequent correction, extensions, or restrictions.

In our case, grammar descriptions will include disambiguation information, so adding disambiguation information is yet another kind of transformation step present in our evolution process.

2.3 Grammar testing

In software testing, a global distinction can be made between white box testing and black box testing. In black box testing, also called functional or behavioural testing, only the external interface of the subject system is available. In white box testing, also called unit testing, the internal composition of the subject system is taken into consideration, and the individual units of this composition can be tested separately.

In grammar testing, we make a similar distinction between functional tests and unit tests. A functional grammar test will use complete VDM-SL specifications as test data. The grammar is tested by generating a parser from it and running this parser on such specifications. Test observations are the success or failure of the parser on a specification, and perhaps its time and space consumption. A unit test will use fragments of VDM-SL specifications as test data. Typically, such fragments are composed

Size and complexity metrics	
TERM	Number of terminals
VAR	Number of non-terminals
MCC	McCabe's cyclometric complexity
AVS	Average size of RHS
Structure metrics	
TIMP	Tree impurity (%)
CLEV	Normalized count of levels (%)
NSLEV	Number of non-singleton levels
DEP	Size of largest level
HEI	Maximum height

Fig. 2. Size and complexity metrics for grammars.

by the grammar developer to help him detect and solve specific errors in the grammar, and to protect himself from reintroducing the error in subsequent development iterations. In addition to success and failure observations, unit tests may observe the number of ambiguities that occur during parsing, or the shape of the parse trees that are produced.

2.4 Grammar metrics

Quantification is an important instrument in understanding and controlling grammar evolution, just as it is for software evolution in general. We have adopted, adapted, and extended the suite of metrics defined for BNF in [20] and implemented a tool, called SdfMetz, to collect grammar metrics for SDF grammars. Here, we will provide a brief description of the metrics we used during the development of the VDM grammar. Full details about the definition and the implementation of these SDF metrics are provided in [2].

SDF differs from (E)BNF in more than syntax. For instance, it allows several productions for the same non-terminal. This forced us to choose between using the number of productions or the number of non-terminals in some metrics definitions. Furthermore, SDF grammars contain more than just context-free syntax. They also contain lexical syntax and disambiguation information. We decided to apply the metrics originally defined for BNF only to the context-free syntax, to make comparisons possible with the results of others. For the disambiguation information a dedicated set of metrics was defined. Thus, we will discuss several categories of metrics: size, complexity, and structure metrics, Halstead metrics, and disambiguation metrics.

Halstead metrics	
n1	Number of distinct operators
n2	Number of distinct opreands
N1	Total number of operators
N2	Total number of operands
n	Program vocabulary
N	Program length
V	Program volume
D	Program difficulty
E	Program effort
L	Program level
T	Program time

Fig. 3. Halstead metrics for grammars

Size, complexity, and structure metrics Figure 2 lists a number of size, complexity, and structure metrics for grammars. These metrics are defined for BNF in [20]. The number of terminals (TERM) and non-terminals (VAR) are simple metrics applicable equally to BNF and SDF. McCabe's cyclometric complexity (MCC), originally defined for program complexity, was adapted for BNF grammars, based on an analogy between grammar production rules and program procedures. Using the same analogy, MCC can be extended easily to cover the operators that SDF adds to BNF.

The average size of right-hand sides (AVS) needs to be adapted to SDF with more care. In (E)BNF the definition of AVS is trivial: count the number of terminals and non-terminals on the right-hand side of each grammar rule, sum these numbers, and divide them by the number of rules. In SDF, this definition can be interpreted in two ways, because each non-terminal can have several productions associated to it. Therefore, we decided to split AVS into two separate metrics: average size of right-hand sides per production (AVS-P) and average size of right-hand sides per non-terminal (AVS-N).

Halstead metrics The Halstead Effort metric [8] has also been adapted for BNF grammars [20]. We will show values not only for Halstead's effort metric but also for some of its ingredient metrics and related metrics. Figure 3 shows a full list. The essential step in adapting Halstead's metrics to grammars is to interpret the notions of *operand* and *operator* in the context of grammars. For details of how we extended this interpretation from BNF to SDF we refer again to [2].

Ambiguity metrics	
FRST	Number of follow restrictions
ASSOC	Number of associativity attributes
REJP	Number of reject productions
UPP	Number of unique productions in priorities
TUPG	Total number of unique productions per group
TNPC	Total number of priority chains
TNG	Total number of priority groups

Fig. 4. Ambiguity metrics for grammars

Ambiguity metrics In SDF, disambiguation constructs are provided in the same formalism as the syntax description itself. To quantify this part of SDF grammars, we defined a series of metrics, which are shown in Figure 4.

2.5 Coverage metrics

To determine how well a given grammar has been tested, a commonly used indicator is the number of non-empty lines in the test suites. We computed these numbers for our functional test suite and unit test suite.

A more reliable instrument to determine grammar test quality is coverage analysis. We have adopted the rule coverage (RC) metric [21] for this purpose. The RC metric simply counts the number of production rules used during parsing of a test suite, and expresses it as a percentage of the total number of production rules of the grammar.

In the case of SDF, several interpretations of RC are possible, due to the fact that a single non-terminal may be defined by multiple productions. One possibility is to count each of these alternative productions separately. Another possibility is to count different productions of the same non-terminal as one. For comparison with rule coverage for BNF grammars, the latter is more appropriate. However, the former gives a more accurate indication of how extensively a grammar is covered by the given test suite. Below we report both under the names of RC (rule coverage) and NC (non-terminal coverage), respectively.

An even more accurate indication can be obtained with context-dependent rule coverage [15]. This metric takes into account not just whether a given production is used, but also whether it has been used in each context (use site) where it can actually occur. However, implementation and computation of this metric is more involved.

3 Development of the VDM grammar

In this section we describe the evolution of the VDM-SL grammar. We describe the release plan and its execution. We provide measurement data on the evolution process and interpretations of the measured values. We describe the test suites used, and the evolution of the unit tests during development.

3.1 Scope and priorities

Language Though we are interested in developing grammars for various existing VDM dialects, such as IFAD VDM and VDM++, we limited the scope of the initial project to the VDM-SL language as describe in the ISO VDM-SL standard [10].

Grammar shape Not only should the parser generate the VDM-SL language exactly as defined in the standard, we also want the shape of the grammar, the names of the non-terminals, and the module structure to correspond closely to the grammar. We want to take advantage of SDF's advanced regular expression-style constructs wherever this leads to additional conciseness and understandability.

Parsing and parse trees Though the grammar should be suitable for generation of a wide range of tool components and tools, we limited the scope of the initial project to development of a grammar from which a GLR parser can be generated. The generated parser should be well-tested, exhibit acceptable time and space consumption, parse without ambiguities, and build abstract syntax trees that correspond as close as possible to the abstract syntax defined in the ISO standard.

3.2 Release plan

Based on the defined scope and priorities, a release plan was drawn up with three releases within the scope of the initial project:

Initial grammar Straightforward transcription of the concrete syntax BNF specification of the ISO standard into SDF notation. Introduction of extended SDF constructs.

Disambiguated grammar Addition of disambiguation information to the grammar, to obtain a grammar from which a non-ambiguous GLR parser can be generated.

Refactored grammar Addition of constructor attributes to context-free productions to allow generated parsers to automatically build ASTs with constructor names corresponding to abstract syntax of the standard. Changes in the grammar shape to better reflect the tree shape as intended by the abstract syntax in the standard.

The following functionalities have explicitly been kept outside the scope of the initial project, and are planned to be added in follow-up projects:

- Haskell front-end[1]. Including generated support for parsing, pretty-printing, AST representation, AST traversal, marshalling ASTs to XML format and back, marshalling of ASTs to ATerm format and back.
- Java front-end. Including similar generated support.
- IFAD VDM-SL extensions, including module syntax.
- VDM object-orientation extension (VDM^{++}).
- Higher test coverage (through test suite development and/or generation).

3.3 Grammar creation and transformation

To accurately keep track of all grammar changes, for each transformation a new revision was created. This led to the creation of a total of 48 development versions. While the first and the latest release versions (initial and refactored) correspond to development versions 1 and 48 of the grammar, respectively, the intermediate release version (disambiguated) corresponds to development version 32.

The initial grammar The grammar was typed from the hardcopy of the ISO Standard [10]. Apart from changing syntax from BNF to SDF, the following was involved in this transcription.

Added SDF constructs Although a direct transcription from EBNF specification was possible, we preferred to use SDF specific constructs. For instance consider the following excerpt from the ISO VDM-SL BNF grammar:

```
product type = type, "*", type, { "*", type} ;
```

During transcription this was converted to:

[1] At the time of writing, a release with this functionality has been completed.

```
{ Type "*" }2+ -> ProductType
```

Both excerpts define the same language. Apart from the syntactic differences from BNF to SDF, the difference is that SDF has special constructs that allows definition of the repetition of a non-terminal separated by a terminal. In this case, the non-terminal `Type` appears at least two times and is always separated by the terminal `"*"`.

Detected top and bottom sorts To help the manual process of typing the grammar a small tool was developed to detect top and bottom sorts. This tool helped to indicate typos required to restore connectedness. More than one top sort, or a bottom sort that is not a lexical (regular-expressions) indicates that a part of the grammar is not connected. This tool provided a valuable help not only in this phase but also during the overall development of the grammar.

Modularized the grammar BNF does not support modularization. The ISO Standard separate concerns by dividing the BNF rules over sections. SDF does support modules, which allowed us to modularize the grammar following the sectioning of the ISO standard.

Added lexical syntax In SDF, lexical syntax can be defined in the same grammar as context-free syntax, using the same notation. In the ISO standard, lexical syntax is described in an adhoc notation resembling BNF, without clear semantics. We interpreted this lexical syntax description and converted it into SDF. Obtaining a complete and correct definition required renaming some lexical non-terminals and providing additionl definitions.

Disambiguation In SDF, disambiguation is specified by means of dedicated disambiguation constructs [4]. These are specified more or less independently from the context-free grammar rules. The constructs are associativity attributes, priorities, reject productions and lookahead restrictions.

In the ISO standard, disambiguation is described in detail by means of tables and a semi-formal textual notation. We interpreted these descriptions and expressed them with SDF disambiguation constructs. This was not a completely straightforward process, in the sense that it is not possible to simply translate the information of the standard document to SDF notation. In some cases, the grammar must respect specific patterns in order enable disambiguation. For each disambiguation specified, a unit test was created.

Origin	LOC	RC	NC
Specification of the MAA standard (Graeme Parkin)	269	19%	30%
Abstract data types (Matthew Suderman and Rick Sutcliffe)	1287	37%	53%
A crosswords assistant (Yves Ledru)	144	28%	43%
Modelling of Realms in (Peter Gorm Larsen)	380	26%	38%
Exercises formal methods course Univ. do Minho (Tiago Alves)	500	35%	48%
Total	2580	50%	70%

Table 1. Integration test suite. The second column gives the number of code lines. The third and fourth columns gives coverage values for the final grammar.

Refactoring As already mentioned, the purpose of this release was to automatically generate ASTs following the ISO standard as close as possible. Two operations were performed:

- added constructor attributes to the contex-free rules
- removed injections to make the AST nicer

The removal of the injections needs further explanation. We call a production rule an injection when it is the only defining production of its non-terminal, and its right-hand side contains exactly one (different) non-terminal. Such injections were actively removed from the grammar, because they needlessly increase the size of the grammar and reduce its readability.

3.4 Test suites

Integration test suite The body of VDM-SL code that strictly adheres to the ISO standard is rather small. Most industrial applications have been developed with tools that support some superset or other deviation from the standard, such as VDM^{++}. We have constructed an integration test suite by collecting specifications from the internet[2]. A preprocessing step was done to extract VDM-SL specification code from literate specifications. We manually adapted specifications that did not adhere to the ISO standard.

Table 1 lists the suite of integration tests that we obtained in this way. The table also shows the lines of code (excluding blank lines) that each test specification contains, as well as the rule coverage (RC) and non-terminal coverage (NC) metrics for each. The coverage metrics shown were obtained with the final, refactored grammar.

[2] A collection of specifications is available from http://www.csr.ncl.ac.uk/vdm/.

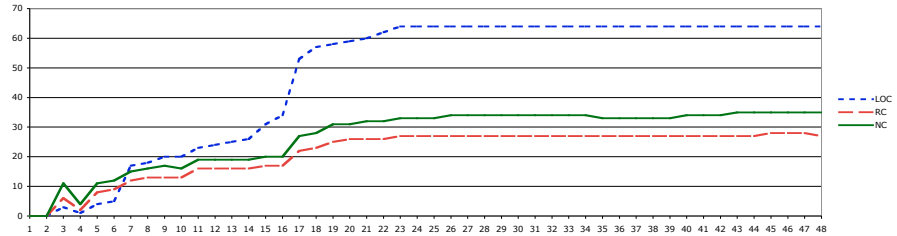

Fig. 5. The evolution of unit tests during development. The x-axis represents the 48 development versions. The three release versions among these are 1, 32, and 48. The left y-axis corresponds to lines of unit test code. Rule coverage (RC) and non-terminal coverage (NC) are shown as well.

Version	TERM	VAR	MCC	AVS-N	AVS-P	HAL	TIMP	CLEV	NSLEV	DEP	HEI
initial	138	161	234	4.4	2.3	55.4	1%	34.9	4	69	16
disambiguated	138	118	232	6.4	2.8	61.1	1.5%	43.9	4	39	16
refactored	138	71	232	10.4	3.3	68.2	3%	52.6	3	27	14

Table 2. Grammar metrics for the three release versions.

Note that in spite of the small size of the integration test suite in terms of lines of code, the test coverage it offers for the grammar is satisfactory. Still, since test coverage is not 100%, a follow-up project specifically aimed at enlarging the integration test suite would be justified.

Unit tests During development, unit tests were created incrementally. For every problem encountered, one or more unit tests were created to isolate the problem.

We measured unit tests development during grammar evolution in terms of lines of unit test code, and coverage by unit tests in terms of rules (RC) and non-terminals (NC). This development is shown graphically in Figure 5. As the chart indicates, all unit tests were developed during the disambiguation phase, i.e. between development versions 1 and 32. There is a small fluctuation in the beginning of the disambiguation process that is due to unit-test strategy changes. Also, between version 23 and 32 unit tests were not added, because the shape of the tree was being tested and this is not covered by our unit-test.

During the refactoring phase, the previously developed unit tests were used to prevent introducing errors unwittingly. Small fluctuations of coverage metrics during this phases are strictly due to variations in the total numbers of production rules and non-terminals.

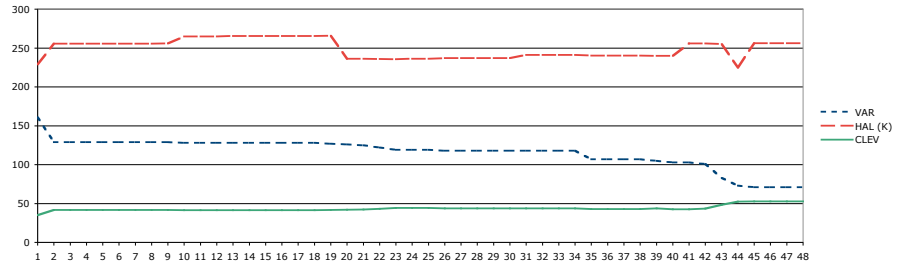

Fig. 6. The evolution of grammar metrics during development. The x-axis represents the 48 development versions.

3.5 Grammar evolution

We measured grammar evolution in terms of the size, complexity, and structure metrics introduced above. This development is summarized in Table 2. This table shows the values of all metrics for the three released versions. In addition, Figure 6 graphically plots the evolution of a selection of the metrics for all 48 development versions.

Size and complexity metrics A first important observation to make is that the number of terminals is constant throughout grammar development. This is conform to expectation, since all keywords and symbols of the language are present from the first grammar version onward.

The initial number of 161 non-terminals decreases via 118 after disambiguation to 71 after refactoring. These numbers are the consequence of changes in grammar shape where non-terminals are replaced by their definition. In the disambiguation phase (43 non-terminals removed), such non-terminal inlining (unfolding) was performed to make formulation of the disambiguation information possible, or easier. For instance, after inlining, simple associativity attributes would suffice to specify disambiguation, while without inlining more elaborate reject productions might have been necessary. In the refactoring phase (47 non-terminals removed), the inlinings performed were mainly removals of injections. These were performed to make the grammar easier to read, more concise, and suitable for creation of ASTs closer to the abstract syntax specification in the standard.

The value of the McCabe cyclometric complexity metric decreases by 2 during disambiguation, meaning that we eliminated two paths in the flow graph of the grammar. This was caused by refactoring the syntax of

product types and union types in similar ways. In case of product types, the following two production rules:

```
ProductType    -> Type
{ Type "*" }2+ -> ProductType
```

were replaced by a single one:

```
Type "*" Type  -> Type
```

For union types, the same replacement was performed. The language generated by the grammar remained the same after these refactorings, but disambiguation became possible.

The average rules size metrics, AVS-N and AVS-P increase significantly. These increases are also due to inlining of non-terminals. Naturally, when a non-terminal with a right-hand size of more than 1 is inlined, the number of non-terminals decreases by 1, and the right-hand sides of the productions in which the non-terminal was used goes up. The increase of AVS-N is roughly by a factor of 2.4, while the increase of AVS-P is by a factor of 1.4. Both the absolute values and the increase of AVS-P are more in accordance with numbers reported in the literature for AVS defined for BNF.

The value of the Halstead Effort metric (HAL) fluctuates during development. It starts at 228K in the initial grammar, and immediately rises to 255K. This initial rise is directly related to the removal of 32 non-terminals. The value then rises more calmly to 265K, but drops again abruptly towards the end of the disambiguation phase, to the level of 236K. During refactoring, the value rises again to 255K, drops briefly to 224K, and finally stabilizes at 256K.

Tree impurity (TIMP) measures how much the grammar's flow graph resembles a tree, expressed as a percentage. The low values for this measure indicates that our grammar is almost a tree, and the dependency complexity is low. As the grammar evolves, the tree impurity increases steadily, from little more than 1%, to little over 3%. This development can be attributed directly to the non-terminal inling that was performed. When a non-terminal is inlined, the flow graph becomes smaller, but the number of cycles remains equal, i.e. the ratio of the latter becomes higher.

Normalized count of levels (CLEV) indicates roughly the percentage of modularizability, if grammar levels (strongly connected components in the flow graph) are considered as modules. Throughout development, the number of levels goes down (from 58 to 40; values are not shown), but the *potential* number of levels, i.e. the number of non-terminals, goes down

Grammar	TERM	VAR	MCC	AVS-N	AVS-P	HAL	TIMP	CLEV	NSLEV	DEP	HEI
Fortran 77	21	16	32	8.8	3.4	**26**	11.7	95.0	1	2	7
ISO C	86	65	149	5.9	5.9	**51**	64.1	33.8	3	38	13
Java v1.1	100	149	213	4.1	4.1	**95**	32.7	59.7	4	33	23
AT&T SDL	83	91	170	5.0	2.6	**138**	1.7	84.8	2	13	15
ISO C^{++}	116	141	368	6.1	6.1	**173**	85.8	14.9	1	121	4
ECMA Standard C$^{\#}$	138	145	466	4.7	4.7	**228**	29.7	64.9	5	44	28
ISO VDM-SL	138	71	232	10.4	3.3	**256**	3.0	52.6	3	27	14
VS Cobol II	333	493	739	3.2	1.9	**306**	0.24	94.4	3	20	27
VS Cobol II (*alt*)	364	185	1158	10.4	8.3	**678**	1.18	82.6	5	21	15
PL/SQL	440	499	888	4.5	2.1	**715**	0.3	87.4	2	38	29

Table 3. Grammar metrics for VDM and other grammars. The italicized grammars are in BNF, and their metrics are reproduced from [20]. The remaining grammars are in SDF. Rows have been sorted by Halstead effort (HAL), which is reported in thousands.

more drastically (from 161 to 71). As a result, CLEV rises from 34% to 53%, meaning that the percentage of modularizability increases.

The number of non-singleton levels (NSLEV) of the grammar is 4 throughout most of its development, except at the end, where it goes down to 3. Inspection of the grammar learns us that these 4 levels roughly correspond to *Expressions*, *Statement*, *Type* and *StateDesignators*. The latter becomes a singleton level towards the end of development due inlining.

The size of the largest grammer level (DEP) starts initially very high at 69 non-terminals, but drops immediately to only 39. Towards the end of development, this number drops further to 27 non-terminals in the largest level.

The height of the level graph (HEI) is 16 throughout most of the evolution of the grammar, but sinks slightly to 14 versus the end of development.

4 Grammar comparisons

In this section we compare our grammar, in terms of metrics, to those developed by others in SDF, and in Yacc-style BNF. The relevant numbers are listed in Table 3, sorted by the value of the Haltstead metric (HAL).

The numbers for the grammars of C, Java, C^{++}, and C$^{\#}$ are reproduced from the same paper from which we adopted the various grammar metrics [20]. These grammars were specified in BNF, or Yacc-like BNF dialects. Note that for these grammars, the AVS-N and AVS-P metrics are always equal, since the number of productions and non-terminals is always equal in BNF grammars.

The numbers for the remaining grammars were computed by us. These grammars were all specified in SDF by various authors. Two versions of the VS Cobol II grammar are listed: the one marked *alt* makes heavy use of nested alternatives, while in the other one, such nested alternatives have been folded into new non-terminals.

Note that the tree impurity (TIMP) values for the SDF grammar are much smaller (between 0.2% and 12%) than for the BNF grammars (between 29% and 86%). This can be attributed to SDF's extended set of regular expression-style constructs, which allow more kinds of iterations to be specified without (mutually) recursive production rules.

In terms of Halstead effort, our VDM-SL grammar ranks quite high, only behind the grammars of the giant Cobol and PL/SQL languages.

5 Concluding remarks

5.1 Contributions

This report documents the process of developing an industrial-strength grammar for the VDM-SL language, taking its ISO standard as starting point. We have monitored the development process with various grammar metrics, and developed unit tests to guide the process of its disambiguation. Apart from the grammar itself, the following contributions were made:

- We extended the collection of metric reference data of [20] with values for 6 additional grammars of widely used languages.
- We collected data for additional grammar metrics defined by us in [2].
- We showed how grammar testing, grammar metrication, and coverage analysis can be combined in systematic grammar development process.

The resulting grammar is readily available from the authors web pages under an open source license. Further development of the grammar takes place in an open, collaborative fashion in the form of a SourceForge project.[3] It is already being used for development of new VDM tools in the context of Formal Methods research [1]. We hope it will be of equal utility to formal method tool initiatives such as the Overture project.[4]

[3] See http://voodoom.sourceforge.net/.
[4] See http://www.overturetool.org/.

5.2 Related work

Malloy and Power have applied various software engineering techniques during the development of a LALR parser for C# [19]. Their techniques include versioning, testing, and the grammar size, complexity, and structure metrics that we adopted ([20], see Table 2). They do not measure coverage, nor do they develop unit tests.

Lämmel et. al. have advocated derivation of grammars from language reference documents through a semi-automatic transformational process [17, 16]. In particular, they have applied their techniques to recover the VS COBOL II grammar from railroad diagrams in an IBM reference manual. They use metrication on grammars, though less extensive than we. Coverage measurement nor unit tests are reported.

Klint et. al. provide an survey over grammar engineering techniques and a agenda for grammar engineering research [13]. In the area of natural language processing, the need for an engineering appraoch and tool support for grammar development has also been recognized [6, 24].

5.3 Future work

Lämmel generalized the notion of rule coverage and advocates the uses of coverage analysis in grammar development [15]. When adopting a transformational approach to grammar completion and correction, coverage analysis can be used to improve grammar testing, and test set generation can be used to increase coverage. SDF tool support for such test set generation and context-dependent rule coverage analysis has yet to be developed.

We plan to extend the grammar in a modular way to cover other dialects of the VDM-SL language, such as IFAD VDM and VDM^{++}. We have already generated Haskell support for VDM processing for the grammar, and are planning to provide generate Java support as well. The integration test suite deserves further extension in order to increase coverage.

5.4 Availability

The final version of the ISO VDM-SL grammar in SDF (development version 48, release version 0.0.3) is included in the appendix below. In addition, this version is available as browseable hyperdocument from http://voodoom.sourceforge.net/iso-vdm.html. All intermediate versions can be obtained from the CVS repository at the project web site at http://voodoom.sourceforge.net/.

References

1. T. Alves, P. Silva, J. Visser, and J.N. Oliveira. Strategic term rewriting and its application to a VDM-SL to SQL conversion. In *Proceedings of the Formal Methods Symposium (FM'05)*. Springer, 2005. To appear.
2. T. Alves and J. Visser. Metrication of SDF grammars. Technical Report DI-PURe-05.05.01, Universidade do Minho, May 2005.
3. M. van den Brand, A. van Deursen, J. Heering, H. de Jonge, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*. Springer-Verlag, 2001.
4. M.G.J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
5. M.G.J. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 108. IEEE Computer Society, 1998.
6. G. Erbach. Tools for grammar engineering, March 15 2000.
7. K.F. Fogel. *Open Source Development with CVS*. Coriolis Group Books, 1999.
8. M.H. Halstead. *Elements of Software Science*, volume 7 of *Operating, and Programming Systems Series*. Elsevier, New York, NY, 1977.
9. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
10. International Organisation for Standardization. *Information technology—Programming languages, their environments and system software interfaces—Vienna Development Method—Specification Language—Part 1: Base language*, December 1996. ISO/IEC 13817-1.
11. M. de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.
12. M. de Jonge and J. Visser. Grammars as contracts. In *Proceedings of the Second International Conference on Generative and Component-based Software Engineering (GCSE 2000)*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2000.
13. P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *Transaction on Software Engineering and Methodology*, 2005. To appear.
14. T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA).
15. R. Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.
16. R. Lämmel. The Amsterdam toolkit for language archaeology (Extended Abstract). In *Proceedings of the 2nd International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering (ATEM 2004)*, October 2004.

17. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.
18. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.
19. B. A. Malloy, J. F. Power, and J. T. Waldron. Applying software engineering techniques to parser design: the development of a C# parser. In *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 75–82. South African Institute for Computer Scientists and Information Technologists, 2002.
20. J.F. Power and B.A. Malloy. A metrics suite for grammar-based software. In *Journal of Software Maintenance and Evolution*, volume 16, pages 405–426. Wiley, November 2004.
21. P. Purdom. Erratum: "A Sentence Generator for Testing Parsers" [BIT **12**(3), 1972, p. 372]. *BIT*, 12(4):595–595, 1972.
22. E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.
23. E. Visser and Z. Benaissa. A Core Language for Rewriting. In C. Kirchner and H. Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *ENTCS*, Pont-à-Mousson, France, September 1998. Elsevier Science.
24. M. Volk. The role of testing in grammar engineering. In *Proceedings of the third conference on Applied natural language processing*, pages 257–258. Association for Computational Linguistics, 1992.

Appendix: ISO VDM-SL Grammar in SDF (final version)

definition

module Main
imports Document

module Document
imports Definitions
exports
  sorts Document
  context-free syntax
    DefinitionBlock+ → Document {**cons**("Document")}

module Definitions
imports TypeDefinitions StateDefinition ValueDefinitions FunctionDefinitions
  OperationDefinitions
exports
  sorts DefinitionBlock
  context-free syntax
    TypeDefinitions → DefinitionBlock {**cons**("DB-TypeDefinitions")}
    StateDefinition → DefinitionBlock {**cons**("DB-StateDefinition")}
    ValueDefinitions → DefinitionBlock {**cons**("DB-ValueDefinitions")}
    FunctionDefinitions → DefinitionBlock {**cons**("DB-FunctionDefinitions")}
    OperationDefinitions → DefinitionBlock {**cons**("DB-OperationDefinitions")}

module TypeDefinitions
imports InvariantDefinition Expressions VDM-Syntax
exports
  sorts TypeDefinitions FieldList Type FunctionType TypeVariableIdentifier
    DiscretionaryType
  context-free syntax
    "types" { TypeDefinition ";"}+ → TypeDefinitions {**cons**("TypeDefinitions")}
    Identifier "=" Type Invariant? → TypeDefinition {**cons**("UnTaggedTypeDef")}
    Identifier "::" FieldList Invariant? → TypeDefinition {**cons**("TaggedTypeDef")}
    "(" Type ")" → Type {**bracket**}
    "bool" → Type {**cons**("BOOL")}
    "nat" → Type {**cons**("NAT")}
    "nat1" → Type {**cons**("NATONE")}
    "int" → Type {**cons**("INTEGER")}
    "rat" → Type {**cons**("RAT")}
    "real" → Type {**cons**("REAL")}
    "char" → Type {**cons**("CHAR")}
    "token" → Type {**cons**("TOKEN")}
    QuoteLiteral → Type {**cons**("QuoteType")}
    "compose" Identifier "of" FieldList "end" → Type {**cons**("CompositeType")}
    Type "|" Type → Type {**left**,**cons**("UnionType")}
    Type "*" Type → Type {**left**,**cons**("ProductType")}
    "[" Type "]" → Type {**cons**("OptionalType")}
    "set" "of" Type → Type {**cons**("SetType")}
    "seq" "of" Type → Type {**cons**("Seq0Type")}
    "seq1" "of" Type → Type {**cons**("Seq1Type")}

```
      "map" Type "to" Type          → Type                  {cons("GeneralMapType")}
      "inmap" Type "to" Type        → Type                  {cons("InjectiveMapType")}
      FunctionType                  → Type                  {cons("FnType")}
      Name                          → Type                  {cons("TypeName")}
      TypeVariableIdentifier        → Type                  {cons("TypeVar")}
      Field*                        → FieldList             {cons("FieldList")}
      (Identifier ":")? Type        → Field                 {cons("Field")}
      DiscretionaryType "->" Type   → FunctionType          {cons("PartialFnType")}
      DiscretionaryType "-t>" Type  → FunctionType          {cons("TotalFnType")}
      Type | ("(" ")")              → DiscretionaryType     {cons("DiscretionaryType")}
      "@" Identifier                → TypeVariableIdentifier {cons("TypeVarId")}
  context-free priorities
    {
      "set" "of" Type           → Type
      "seq" "of" Type           → Type
      "seq1" "of" Type          → Type
      "map" Type "to" Type      → Type
      "inmap" Type "to" Type    → Type
    } >
    {
      Type "*" Type             → Type
    } >
    {
      Type "|" Type             → Type
    } >
    {
      FunctionType              → Type
    }

module InvariantDefinition
imports Expressions PatternsAndBindings
exports
  sorts Invariant InvariantInitialFunction
context-free syntax
    "inv" InvariantInitialFunction → Invariant                {cons("Invariant")}
    Pattern "==" Expression        → InvariantInitialFunction {cons("InvInitFn")}

module StateDefinition
imports TypeDefinitions InvariantDefinition VDM-Syntax
exports
  sorts StateDefinition
  context-free syntax

"state" Identifier "of" FieldList Invariant? Initialization? "end" → StateDefinition {cons("StateDef")}
"init" InvariantInitialFunction                                   → Initialization    {cons("Initialization")}

module ValueDefinitions
imports TypeDefinitions Expressions PatternsAndBindings
exports
  sorts ValueDefinitions ValueDefinition
  context-free syntax
    "values" { ValueDefinition ";" }+    → ValueDefinitions {cons("ValueDefinitions")}
    Pattern (":" Type)? "=" Expression → ValueDefinition  {cons("ValueDef")}

module FunctionDefinitions
imports TypeDefinitions FunctionDefinitions Expressions PatternsAndBindings
  VDM-Syntax
```

```
exports
  sorts FunctionDefinitions Parameters ParameterTypes IdentifierTypePair
    FunctionDefinition
  context-free syntax

"functions" { FunctionDefinition ";" }+ → FunctionDefinitions {cons("FunctionDefinitions")}

Identifier TypeVariableList? ":" FunctionType Identifier
  ParametersList "==" Expression ("pre" Expression)?
                                  → FunctionDefinition  {cons("ExplFnDef")}
Identifier TypeVariableList? ParameterTypes IdentifierTypePair
  ("pre" Expression)? "post" Expression
                                  → FunctionDefinition  {cons("ImplFnDef")}
"[" { TypeVariableIdentifier "," }+ "]"  → TypeVariableList   {cons("TypeVarList")}
    Identifier ":" Type             → IdentifierTypePair    {cons("IdType")}
    "(" PatternTypePairList? ")"    → ParameterTypes        {cons("ParameterTypes")}
    { PatternTypePair "," }+        → PatternTypePairList   {cons("PatTypePairList")}
    PatternList ":" Type            → PatternTypePair       {cons("PatTypePair")}
    Parameters+                     → ParametersList        {cons("ParametersList")}
    "(" PatternList? ")"            → Parameters            {cons("Parameters")}


module OperationDefinitions
imports TypeDefinitions FunctionDefinitions Expressions Statements VDM-Syntax
exports
  sorts OperationDefinitions
  context-free syntax
    "operations" { OperationDefinition ";" }+     → OperationDefinitions {cons("OperationDefinitions")}
    Identifier ":" OperationType Identifier
      Parameters "==" Statement ("pre" Expression)?
                                  → OperationDefinition  {cons("ExplOprtDef")}
    Identifier ParameterTypes IdentifierTypePair? Externals?
      ("pre" Expression)? "post" Expression Exceptions?
                                  → OperationDefinition  {cons("ImplOprtDef")}
    DiscretionaryType "==>" DiscretionaryType → OperationType   {cons("OpType")}
    "ext" VarInformation+         → Externals               {cons("Externals")}
    Mode NameList (":" Type)?     → VarInformation          {cons("VarInf")}
    "rd"                          → Mode                    {cons("READ")}
    "wr"                          → Mode                    {cons("READWRITE")}
    "errs" Error+                 → Exceptions              {cons("Exceptions")}
    Identifier ":" Expression "−>" Expression  → Error      {cons("Error")}


module Expressions
imports TypeDefinitions Statements PatternsAndBindings VDM-Syntax
exports
  sorts ExpressionList Expression NameList Name Maplet
  context-free syntax

{ Expression "," }+                                  → ExpressionList {cons("ExprList")}
"(" Expression ")"                                   → Expression     {bracket}
"let" { LocalDefinition "," }+ "in" Expression       → Expression     {cons("LetExpr")}
"let" Bind ("be" "st" Expression)? "in" Expression   → Expression     {cons("LetBeSTExpr")}
"def" { DefBind ";" }+ "in" Expression               → Expression     {cons("DefExpr")}
PatternBind "=" Expression                           → DefBind        {cons("DefBind")}
    "if" Expression "then" Expression ElseIfExpression* "else" Expression
                                  → Expression       {cons("IfExpr")}
    "elseif" Expression "then" Expression → ElseIfExpression {cons("ElsifExpr")}
```

“cases” Expression “:” CasesExpressionAlternatives (“,” OthersExpression)? “end”	→ Expression	{**cons**(“CasesExpr”)}
{ CasesExpressionAlternative “,”}+	→ CasesExpressionAlternatives	{**cons**(“CasesExprAlternatives”)}
PatternList “−>” Expression	→ CasesExpressionAlternative	{**cons**(“CaseAltn”)}
“others” “−>” Expression	→ OthersExpression	{**cons**(“OthersExpr”)}

```
"+" Expression                  → Expression {cons("UnaryNumPlusExpr")}
"−" Expression                  → Expression {cons("UnaryNumMinusExpr")}
"abs" Expression                → Expression {cons("NumAbsExpr")}
"floor" Expression              → Expression {cons("FloorExpr")}
"not" Expression                → Expression {cons("NotExpr")}
"card" Expression               → Expression {cons("SetCardExpr")}
"power" Expression              → Expression {cons("SetPowerExpr")}
"dunion" Expression             → Expression {cons("SetDistrUnionExpr")}
"dinter" Expression             → Expression {cons("SetDistrIntersectExpr")}
"hd" Expression                 → Expression {cons("SeqHeadExpr")}
"tl" Expression                 → Expression {cons("SeqTailExpr")}
"len" Expression                → Expression {cons("SeqLenExpr")}
"elems" Expression              → Expression {cons("SeqElemsExpr")}
"inds" Expression               → Expression {cons("SeqIndicesExpr")}
"conc" Expression               → Expression {cons("SeqDistrConcExpr")}
"dom" Expression                → Expression {cons("MapDomExpr")}
"rng" Expression                → Expression {cons("MapRngExpr")}
"merge" Expression              → Expression {cons("MapDistrMergeExpr")}
"inverse" Expression            → Expression {cons("MapInverseExpr")}
Expression "+" Expression       → Expression {left,cons("BinaryNumPlusExpr")}
Expression "−" Expression       → Expression {left,cons("BinaryNumMinusExpr")}
Expression "*" Expression       → Expression {left,cons("NumPlusExpr")}
Expression "/" Expression       → Expression {left,cons("NumDivExpr")}
Expression "div" Expression     → Expression {left,cons("IntDivExpr")}
Expression "rem" Expression     → Expression {left,cons("NumRemExpr")}
Expression "mod" Expression     → Expression {left,cons("NumModExpr")}
Expression "<" Expression       → Expression {left,cons("NumLtExpr")}
Expression "<=" Expression      → Expression {left,cons("NumLeExpr")}
Expression ">" Expression       → Expression {left,cons("NumGtExpr")}
Expression ">=" Expression      → Expression {left,cons("NumGeExpr")}
Expression "=" Expression       → Expression {left,cons("EqExpr")}
Expression "<>" Expression      → Expression {left,cons("NeExpr")}
Expression "or" Expression      → Expression {assoc,cons("OrExpr")}
Expression "and" Expression     → Expression {assoc,cons("AndExpr")}
Expression "=>" Expression      → Expression {right,cons("ImplyExpr")}
Expression "<=>" Expression     → Expression {assoc,cons("EquivExpr")}
Expression ("in" "set") Expression          → Expression {left,cons("InSetExpr")}
Expression ("not" "in" "set") Expression    → Expression {left,cons("NotInSetExpr")}
Expression "subset" Expression              → Expression {left,cons("SubsetExpr")}
Expression "psubset" Expression             → Expression {left,cons("ProperSubsetExpr")}
Expression "union" Expression               → Expression {left,cons("SetUnionExpr")}
Expression "\\" Expression                  → Expression {left,cons("SetDifference")}
Expression "inter" Expression               → Expression {left,cons("SetIntersect")}
Expression "^" Expression                   → Expression {left,cons("SeqConc")}
Expression "++" Expression      → Expression {left,prefer,cons("MapOrSeqModExpr")}
Expression "munion" Expression  → Expression {left,cons("MapMergeExpr")}
Expression "<:" Expression      → Expression {left,cons("MapDomRestrToExpr")}
Expression "<−:" Expression     → Expression {left,cons("MapDomRestrByExpr")}
Expression ":>" Expression      → Expression {left,cons("MapRngRestrToExpr")}
Expression ":−>" Expression     → Expression {left,cons("MapRngRestrByExpr")}
Expression "comp" Expression    → Expression {right,cons("ComposeExpr")}
Expression "**" Expression      → Expression {right,cons("IterateExpr")}
QuantifiedExpression                    → Expression           {cons("QuantExpr")}
"forall" BindList "&" Expression        → QuantifiedExpression {cons("AllExpr")}
"exists" BindList "&" Expression        → QuantifiedExpression {cons("ExistsExpr")}
"exists1" BindList "&" Expression       → QuantifiedExpression {cons("ExistsUniqueExpr")}
```

```
    "iota" Bind "&" Expression          → Expression {cons("IotaExpr")}
    "{" ExpressionList? "}"             → Expression {cons("SetEnumeration")}
    "{" Expression "|" BindList ("&" Expression)? "}"
                                        → Expression {cons("SetComprehension")}
    "{" Expression "," "..." "," Expression "}"
                                        → Expression {cons("SetRange")}
    "[" ExpressionList? "]"             → Expression {cons("SeqEnumeration")}
    "[" Expression "|" SetBind ("&" Expression)? "]"
                                        → Expression {cons("SeqComprehension")}
    Expression "(" Expression "," "..." "," Expression ")"
                                        → Expression {cons("SubSequence")}
    ("{" "|->" "}") | ("{" { Maplet "," }+ "}")
                                        → Expression {cons("MapEnumeration")}
    "{" Maplet "|" BindList ("&" Expression)? "}"
                                        → Expression {cons("MapComprehension")}
    Expression "|->" Expression         → Maplet     {cons("Maplet")}
    "mk_" "(" Expression "," ExpressionList ")"
                                        → Expression {cons("TupleConstructor")}
    "mk_" Name "(" ExpressionList? ")"
                                        → Expression {cons("RecordConstructor")}
    "mu" "(" Expression "," { RecordModification "," }+ ")"
                                        → Expression {cons("RecordModifier")}
    Identifier "|->" Expression   → RecordModification {cons("RecordModification")}
    Expression "(" ExpressionList? ")" → Expression {cons("Apply")}
    Expression "." Identifier           → Expression {cons("FieldSelect")}
    Name "[" { Type "," }+ "]"          → Expression {cons("FctTypeInst")}
    "lambda" TypeBindList "&" Expression
                                        → Expression {cons("Lambda")}
    "is_" "bool" "(" Expression ")"     → Expression {cons("IsBoolExpr")}
    "is_" "nat" "(" Expression ")"      → Expression {cons("IsNatExpr")}
    "is_" "nat1" "(" Expression ")"     → Expression {cons("IsNatOneExpr")}
    "is_" "int" "(" Expression ")"      → Expression {cons("IsIntegerExpr")}
    "is_" "rat" "(" Expression ")"      → Expression {cons("IsRatExpr")}
    "is_" "real" "(" Expression ")"     → Expression {cons("IsRealExpr")}
    "is_" "char" "(" Expression ")"     → Expression {cons("IsCharExpr")}
    "is_" "token" "(" Expression ")"    → Expression {cons("IsTokenExpr")}
    "is_" Identifier "(" Expression ")" → Expression {cons("IsDefTypeExpr")}
    Name                                → Expression {cons("NameExpr")}
    OldName                             → Expression {cons("OldNameExpr")}
    SymbolicLiteral                     → Expression {cons("LiteralExpr")}
  Identifier        → Name     {cons("Name")}
  { Name "," }+ → NameList {cons("NameList")}
  Identifier " "    → OldName {cons("OldName")}
context-free priorities
  {right:
    Expression "**" Expression     → Expression
    Expression "comp" Expression → Expression
  }>
  {
    Expression "(" Expression "," "..." "," Expression ")" → Expression
    Expression "(" ExpressionList? ")"                     → Expression
    Expression "." Identifier                              → Expression
  }>
  {
```

```
    "+" Expression       → Expression
    "−" Expression       → Expression
    "abs" Expression     → Expression
    "floor" Expression   → Expression
    "card" Expression    → Expression
    "power" Expression   → Expression
    "dunion" Expression  → Expression
    "dinter" Expression  → Expression
    "hd" Expression      → Expression
    "tl" Expression      → Expression
    "len" Expression     → Expression
    "elems" Expression   → Expression
    "inds" Expression    → Expression
    "conc" Expression    → Expression
    "dom" Expression     → Expression
    "rng" Expression     → Expression
    "merge" Expression   → Expression
}>
{left:
    Expression "<:" Expression   → Expression
    Expression "<−:" Expression  → Expression
    Expression ":>" Expression   → Expression
    Expression ":−>" Expression  → Expression
}>
{
    "inverse" Expression → Expression
}>
{left:
    Expression "*" Expression    → Expression
    Expression "/" Expression    → Expression
    Expression "div" Expression  → Expression
    Expression "rem" Expression  → Expression
    Expression "mod" Expression  → Expression
    Expression "inter" Expression → Expression
}>
{left:
    Expression "+" Expression       → Expression
    Expression "−" Expression       → Expression
    Expression "union" Expression   → Expression
    Expression "\\" Expression      → Expression
    Expression "munion" Expression  → Expression
    Expression "++" Expression      → Expression
    Expression "^" Expression       → Expression
}>
{left:
    Expression "<" Expression                  → Expression
    Expression "<=" Expression                 → Expression
    Expression ">" Expression                  → Expression
    Expression ">=" Expression                 → Expression
    Expression "=" Expression                  → Expression
    Expression "<>" Expression                 → Expression
    Expression ("in" "set") Expression         → Expression
    Expression ("not" "in" "set") Expression   → Expression
    Expression "subset" Expression             → Expression
    Expression "psubset" Expression            → Expression
}>
```

```
    "not" Expression                → Expression >
    Expression "and" Expression     → Expression >
    Expression "or" Expression      → Expression >
    Expression "=>" Expression      → Expression >
    Expression "<=>" Expression     → Expression >
    {
      "let" { LocalDefinition "," }+ "in" Expression        → Expression
      "let" Bind ("be" "st" Expression)? "in" Expression    → Expression
      "if" Expression "then" Expression ElseIfExpression* "else" Expression
                                                            → Expression
      "def" { DefBind ";" }+ "in" Expression                → Expression
      "forall" BindList "&" Expression                      → QuantifiedExpression
      "exists" BindList "&" Expression                      → QuantifiedExpression
      "exists1" BindList "&" Expression                     → QuantifiedExpression
      "iota" Bind "&" Expression                            → Expression
      "lambda" TypeBindList "&" Expression                  → Expression
    }

module StateDesignators
imports Expressions VDM-Syntax
exports
  sorts StateDesignator
  context-free syntax
    Name                              → StateDesignator {cons("StDesignatorName")}
    StateDesignator "." Identifier    → StateDesignator {cons("FieldRef")}
    StateDesignator "(" Expression ")" → StateDesignator {cons("MapOrSeqRef")}

module Statements
imports TypeDefinitions ValueDefinitions FunctionDefinitions Expressions StateDesignators
PatternsAndBindings VDM-Syntax
exports
  sorts Statement LocalDefinition
  context-free syntax
    "let" { LocalDefinition "," }+ "in" Statement       → Statement              {cons("LetStmt")}
    FunctionDefinition | ValueDefinition                → LocalDefinition        {cons("LocalDefinition")}
    "let" Bind ( "be" "st" Expression )? "in" Statement → Statement              {cons("LetBeSTStmt")}
    "def" { EqualsDefinition ";" }+ "in" Statement      → Statement              {cons("DefStmt")}
    PatternBind "=" Expression                          → EqualsDefinition       {prefer,cons("EqualsExprDef")}
    PatternBind "=" CallStatement                       → EqualsDefinition       {cons("EqualsCallStmtDef")}
    "(" DclStatement* { Statement ";" }+ ")"            → Statement              {cons("BlockStmt")}
    "dcl" AssignmentDefinition ";"                      → DclStatement           {cons("DclStmt")}
    Identifier ":" Type (":=" Expression)?              → AssignmentDefinition   {prefer,cons("AssignDefExpr")}
    Identifier ":" Type (":=" CallStatement)?           → AssignmentDefinition   {cons("AssignDefCallStmt")}
    AssignStatement                                     → Statement              {cons("AssignStmt")}
    StateDesignator ":=" Expression                     → AssignStatement        {prefer,cons("AssignStmtExpr")}
    StateDesignator ":=" CallStatement                  → AssignStatement        {cons("AssignStmtCallStmt")}
    "if" Expression "then" Statement ElseifStatement* ("else" Statement)?
                                                        → Statement              {cons("IfStmt")}
    "elseif" Expression "then" Statement                → ElseifStatement        {cons("ElsifStmt")}
    "cases" Expression ":" CasesStatementAlternatives ("," OthersStatement)? "end"
                                                        → Statement              {cons("CasesStmt")}
    { CasesStatementAlternative "," }+
        → CasesStatementAlternatives {cons("CasesStmtAlternatives")}
    PatternList "->" Statement                          → CasesStatementAlternative {cons("CasesStmtAltn")}
```

```
"others" "->" Statement                  → OthersStatement {cons("OthersStmt")}
"for" PatternBind "in" "reverse"? Expression "do" Statement
                                         → Statement       {cons("SeqForLoop")}
"for" "all" Pattern "in" "set" Expression "do" Statement
                                         → Statement       {cons("SetForLoop")}
"for" Identifier "=" Expression "to" Expression ("by" Expression)? "do" Statement
                                         → Statement       {cons("IndexForLoop")}
"while" Expression "do" Statement        → Statement       {cons("WhileLoop")}
"||" "(" { Statement "," }+ ")"          → Statement       {cons("NonDetStmt")}
CallStatement                            → Statement       {cons("CallStmt")}
Name "(" ExpressionList? ")" ("using" StateDesignator)?
                                         → CallStatement   {cons("Call")}
"return" Expression?                     → Statement       {cons("ReturnStmt")}
"always" Statement "in" Statement        → Statement       {cons("AlwaysStmt")}
"trap" PatternBind "with" Statement "in" Statement
                                         → Statement       {cons("TrapStmt")}
"tixe" Traps "in" Statement              → Statement       {cons("RecTrapStmt")}
"{" {Trap ","}+ "}"                      → Traps           {cons("Traps")}
PatternBind "|->" Statement              → Trap            {cons("Trap")}
"exit" Expression?                       → Statement       {cons("ExitStmt")}
"skip"                                   → Statement       {cons("IdentStmt")}
```

```
module PatternsAndBindings
imports TypeDefinitions Expressions VDM-Syntax
exports
  sorts Pattern PatternList Bind PatternBind BindList SetBind TypeBindList
  context-free syntax
    "-" | Identifier                     → Pattern {cons("PatternId")}
    "(" Expression ")"                   → Pattern {cons("MatchValueExpr")}
    SymbolicLiteral                      → Pattern {cons("MatchValueSymb")}
    "{" PatternList "}"                  → Pattern {cons("SetEnumPattern")}
    Pattern "union" Pattern              → Pattern {cons("SetUnionPattern")}
    "[" PatternList "]"                  → Pattern {cons("SeqEnumPattern")}
    Pattern "^" Pattern                  → Pattern {cons("SeqConcPattern")}
    "mk_" "(" Pattern "," PatternList ")" → Pattern {cons("TuplePattern")}
    "mk_" Name "(" PatternList? ")"      → Pattern {cons("RecordPattern")}
    { Pattern "," }+                     → PatternList   {cons("PatList")}
    Pattern | Bind                       → PatternBind   {cons("PatternBind")}
    SetBind | TypeBind                   → Bind          {cons("Bind")}
    Pattern "in" "set" Expression        → SetBind       {cons("SetBind")}
    Pattern ":" Type                     → TypeBind      {cons("TypeBind")}
    { MultipleBind "," }+                → BindList      {cons("BindList")}
    PatternList "in" "set" Expression    → MultipleBind  {cons("MultSetBind")}
    PatternList ":" Type                 → MultipleBind  {cons("MultTypeBind")}
    { TypeBind "," }+                    → TypeBindList  {cons("TypeBindList")}
```

```
module VDM-Syntax imports Characters Layout
exports
  sorts SymbolicLiteral Identifier QuoteLiteral
  context-free syntax
    NumericLiteral   → SymbolicLiteral {cons("NumericLiteral")}
    BooleanLiteral   → SymbolicLiteral {cons("BooleanLiteral")}
    "nil"            → SymbolicLiteral {cons("NilLiteral")}
    CharacterLiteral → SymbolicLiteral {cons("CharacterLiteral")}
    TextLiteral      → SymbolicLiteral {cons("TextLiteral")}
    QuoteLiteral     → SymbolicLiteral {cons("QuoteLiteral")}
    "true"           → BooleanLiteral  {cons("TRUE")}
    "false"          → BooleanLiteral  {cons("FALSE")}
```

lexical syntax
Numeral ("." Digit+)? Exponent? → NumericLiteral
Digit+ → Numeral
"E" ("+" | "−")? Numeral → Exponent
"'" Character "'" → CharacterLiteral
"`" Character* "`" → TextLiteral
"<" PlainLetter (PlainLetter | Digit | "'" | "_")* ">" → QuoteLiteral
(PlainLetter | GreekLetter) (PlainLetter | GreekLetter | Digit | "'" | "_")* → Identifier
context-free restrictions
Identifier −/− [a-zA-Z0-9]
NumericLiteral −/− [0-9]
context-free syntax

"abs" | "all" | "always" | "be" | "by" | "card" | "cases" | "char" | "compose" |
"conc" | "dcl" | "def" | "div" | "do" | "dom" | "elems" | "else" | "elseif" | "end" |
"errs" | "exit" | "ext" | "false" | "floor" | "for" | "functions" | "hd" | "if" | "in" |
"inds" | "init" | "inv" | "len" | "let" | "merge" | "mod" | "nil" | "of" | "operations" |
"others" | "post" | "pre" | "rd" | "rem" | "return" | "reverse" | "rng" | "skip" | "st" |
"state" | "then" | "tixe" | "token" | "tl" | "to" | "trap" | "true" | "types" | "using" |
"values" | "while" | "with" | "wr" | "abs" | "card" | "conc" | "dinter" | "dom" |
"dunion" | "elems" | "floor" | "hd" | "inds" | "inverse" | "len" | "merge" | "not" | "power" |
"rng" | "tl" | "mk_" | "mk_" Identifier | "is_" Identifier | "is_" "bool" | "is_" "nat" |
"is_" "nat1" | "is_" "int" | "is_" "rat" | "is_" "real" | "is_" "char" | "is_" "token" |
"bool" | "nat" | "nat1" | "int" | "rat" | "real" | "char" | "token" → Identifier {**reject**}
restrictions
"abs" "all" "always" "and" "be" "bool" "by" "card" "cases" "char"
"compose" "conc" "dcl" "def" "div" "do" "dom" "elems" "else" "elseif"
"end" "errs" "exit" "ext" "false" "floor" "for" "functions" "hd" "if"
"in" "inds" "init" "inv" "len" "let" "merge" "mod" "nil" "of"
"operations" "others" "post" "pre" "rd" "rem" "return" "reverse" "rng"
"skip" "st" "state" "then" "tixe" "token" "tl" "to" "trap" "true" "types"
"using" "values" "while" "with" "wr" "abs" "card" "conc" "dinter" "dom"
"dunion" "elems" "floor" "hd" "inds" "inverse" "len" "merge" "not"
"power" "rng" "tl" "bool" "nat" "nat1" "int" "rat" "real" "char"
"token" −/− [a-zA-Z]

module Characters
exports
sorts Character PlainLetter GreekLetter Digit
lexical syntax

PlainLetter | GreekLetter | Digit | DelimiterCharacter | OtherCharacter | Separator → Character
[a-zA-Z] → PlainLetter
"#" (LowerGreekLetter | UpperGreekLetter) → GreekLetter
[ABGDEZHQIKLMNXOPRSTUFCYW] → UpperGreekLetter
[abgdezhqiklmnxoprstufcyw] → LowerGreekLetter
[0-9] → Digit
"," | ":" | ";" | "=" | "(" | ")" | "|" | "−" | "[" | "]" | "{" | "}" | "+" | "/" | "<" | ">" |
"<=" | ">=" | "<>" | "~" | "forall" | "exists" | "." | "seq" "of" | "seq1" "of" | "inmap" |
"map" | "−>" | "−*t*>" | "==>" | "||" | "=>" | "<=>" | "in" "set" | "|−>" | "dinter" |
"dunion" | "inverse" | "<:" | ":>" | "<−:" | ":−>" | "iota" | "lambda" | "mu" | "&" | "*" |
"==" | "not" | "inter" | "union" | "munion" | "**" | "psubset" | "subset" | "power" |
"not" "in" "set" | "^" | "++" | "comp" | "and" | "or" | "bool" | "nat" | "nat1" | "int" | "rat" |
"real" → DelimiterCharacter
"_" | "'" | "\"" | "@" | "!" | "`" → OtherCharacter
"\t" | "\n" | " " → Separator

```
module Layout
exports
  lexical syntax
    "--"~[\n\13]*[\13\n] → LAYOUT
    [\ \t\n\13]          → LAYOUT
  context-free restrictions
    LAYOUT? −/− [\ \t\n\13]
```