Grammar-centered Development of VDM Support

Tiago Alves* and Joost Visser**

Departamento de Informática, Universidade do Minho, Braga, Portugal
{tiago.alves,joost.visser}@di.uminho.pt
http://www.di.uminho.pt

Abstract. Starting from the ISO language reference, we have developed an industrial strength grammar for the VDM specification language. We summarize both the development process and its result. The employed methodology can be described as iterative grammar engineering and includes 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. In particular, we have generated Haskell support for parsing VDM, traversing the resulting ASTs, representing the ASTs in XML and in the ATerm maximal sharing interchange format, and for pretty-printing the ASTs back to VDM's surface syntax. This front-end has proven its usefulness in the implementation of VooDooM, a tool that supports generation of relational models from VDM data types.

1 Introduction

We advocate a grammar-centered approach to language tool development [16], in which a variety of tool components and support libraries are generated from platform-neutral grammars. We are applying this approach for the development of tool support for the VDM specification language (VDM-SL). In the current paper, we provide a summary of the first two phases of this development.

In the first phase, we have applied grammar engineering techniques to obtain a complete grammar of the VDM-SL language from its ISO standard language reference. This phase is summarized in Sections 2 and 3. A fully detailed treatment can be found in [2], which includes the resulting platform-independent grammar as an appendix.

In the second phase, we have generated a parser, a pretty-printer, and libraries for serialization and traversal of abstract syntax trees from our VDM-SL grammar. We have used these generated components and libraries in the development of VooDooM. This tool converts VDM datatype definitions into the

* Supported by Information Knowledge Fusion, IKF-P E!2235.
** Supported by Fundação para a Ciência e a Tecnologia, SFRH/BPD/11609/2002.

form of a relational model, and was implemented in Haskell. In Section 4, we discuss generation of components and libraries, specifically for target languages Java and Haskell, and we provide some details of how tool construction proceeds after such generation. Full detailes on VooDooM are provided in [1], including the refinement theory that underpins the conversion it implements.

The paper is concluded in Section 5 with a note on availabilty and identification of future challenges. For discussions of related work, we refer to [3] and [1].

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. Through their adoption, the notoriously erratic and unpredictable process of developing and maintaining large grammars can become more efficient and effective, and can lead to results of higher-quality. Such timely delivery of high-quality grammars is especially important in the context of grammar-centered language tool development, where grammars are used for much more than single-platform parser generation.

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 [6].

For the development of language tool support, we advocate a *grammar-centered* approach [16]. 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 can serve as input for generating 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 deserialization in various formats, customizable 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

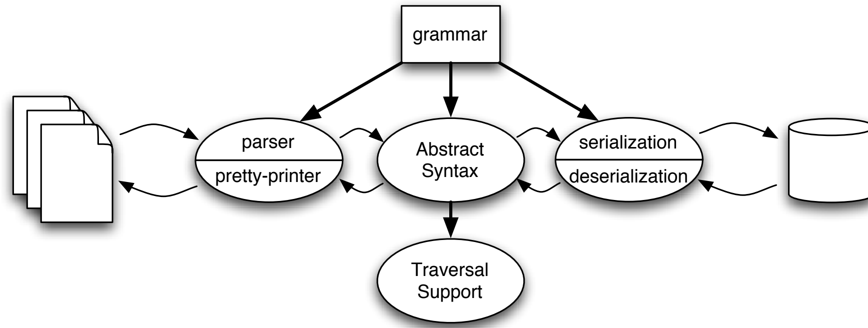

Fig. 1. Grammar-centered approach to language tool development.

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 notation of choice: SDF [12, 22].

The syntax definition formalism SDF allows description of both lexical and context-free syntax. It adds even more regular expression-style constructs 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 [4, 23, 19, 17, 15, 14].

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.

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 [10].

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, or by converting or transcribing a language reference manual, available as electronic or paper document.

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, changing neither the language that is generated, nor its semantics.

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 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. Full details about the definition and the implementation of these SDF metrics are provided in [3]. Here we will provide just a brief description.

Size and complexity metrics Table 1 (left side) lists a number of size and complexity metrics for grammars. McCabe's cyclometric complexity (MCC), originally defined for program complexity, was adapted for grammars, based on an analogy between grammar production rules and program procedures. In SDF, each non-terminal can have several productions associated to it. Therefore, average right hand side (AVS) is split 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). While the AVS-N metric is more appropriate to compare with other formalisms (like BNF), the AVS-P metric is more accurate.

Structure metrics Table 1 (right side) lists a number of structure metrics also previously defined in [20]. The grammar is first represented as a graph that has non-terminal as nodes, and contains edges between two non-terminals whenever one occurs in the RHS of the definition of the other. Only the tree impurity metric (TIMP) is calculated directly from this graph, all the others are calculated from the strongly connected components graph. This graph is obtained from the previous graph in which each node (level) is obtained by grouping the elements that are strongly connected.

Table 1. Size, complexity, and structure metrics for grammars.

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

Tree impurity (TIMP) measures how much the graph resembles a tree, expressed as a percentage. A tree impurity of 0 percent means that the graph is a tree, and a tree impurity of 100 percent means that it is a fully connected graph.

Halstead metrics The Halstead Effort metric [11] has also been adapted for BNF grammars [20]. The essential step in adapting Halstead's metrics to grammars is to interpret the notions of *operand* and *operator* in the context of 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. One of these is the number of unique productions in priorities (UPP).

2.4 Grammar testing

In grammar testing, as in general software testing, a global distinction can be made between functional tests (black box) and unit tests (white box). A functional grammar test will use complete files as test data. A unit test will use fragments of files as test data. Typically, such fragments are composed 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. For both functional and unit testing we have used the `parse-unit` utility [7].

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. 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. To computed these numbers for our functional test suite and unit test suite we developed a tool called SdfCoverage.

SDF allows two possible interpretations of RC, due to the fact that a single non-terminal may be defined by multiple productions. Thus, as in the case of AVS, we measured two metrics, by the names of RC (rule coverage) and NC (non-terminal coverage), respectively.

3 Development of the VDM grammar

We have applied the grammar engineering techniques described above during the iterative development of an SDF grammar of VDM-SL. In this section we first

describe the scope, priorities, and planned deliverables of the project. Then, the evolution of the grammar during development is explained. We provide measurement data on the evolution process and interpretations of the measured values. Finally we describe the test suites used, and the evolution of the unit tests and test coverage during development.

3.1 Scope, priorities, and planned deliverables

We limited the scope of the initial project to the VDM-SL language as described in the ISO VDM-SL standard [13]. Not only should the parser accept 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 standard.

A release plan was drawn up with three releases within the scope of the initial phase of the 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.

3.2 Grammar creation and evolution

To accurately keep track of all grammar changes, for each transformation a new revision was created, leading 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 [13]. In that document, context-free syntax, lexical syntax and disambiguation information is specified in a semi-formal notation. Context-free syntax is specified in EBNF[1], but the terminals are specified as mathematical symbols. To translate the mathematical symbols to ASCII symbols, an interchange table is defined. Lexical syntax is specified in tables by enumerating the possible symbols. Finally, disambiguation information is specified in terms of precedence in tables and equations.

[1] Although the grammar is specified in EBNF only BNF constructs were used.

Apart from changing syntax from EBNF to SDF and using the interchange table to substitute mathematical symbols for their parseable representation, several issues were addressed. For instance, SDF's constructs for iteration (with separators) were introduced to make the grammar more elegant and concise. SDF's modularization features were used to modularize the grammar following the sectioning of the ISO standard. In the ISO standard, lexical syntax is described in an ad-hoc notation, resembling BNF but without clear semantics, plus a table that enumerates symbols. We interpreted this lexical syntax description and converted it into SDF, where lexical syntax can be defined in the same grammar as context-free syntax. Obtaining a complete and correct definition required renaming some lexical non-terminals and providing additional definitions.

Disambiguation In SDF, disambiguation is specified by means of dedicated disambiguation constructs [5]. 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.

Refactoring As already mentioned, the purpose of this release was to automatically generate ASTs following the ISO standard as close as possible. To this end, two operations were performed: addition of constructor attributes to the context-free rules, and removal of injections[2] to make the AST nicer.

3.3 Grammar metrics

We measured grammar evolution in terms of the size, complexity, structure and Halstead metrics introduced above. This development is summarized in Table 2. This table shows the values of all metrics for the three released versions. In

[2] 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.

Table 2. Grammar metrics for the three release versions.

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

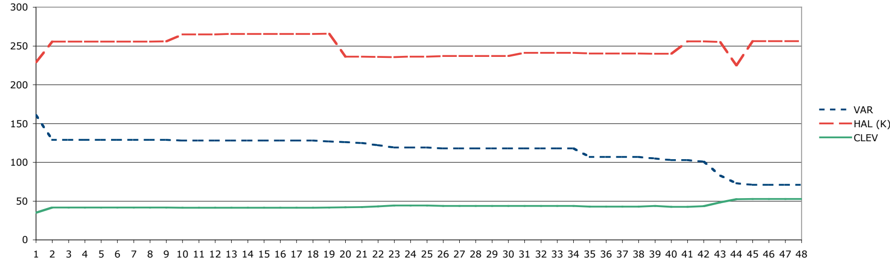

Fig. 2. The evolution of grammar metrics during development. The x-axis represents the 48 development versions.

addition, Figure 2 graphically plots the evolution of a selection of the metrics for all 48 development versions.

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

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

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.

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

Grammar comparisons We have compared 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 Halstead effort metric (HAL). In terms of Halstead effort, our VDM-SL grammar ranks quite high, only behind the grammars of the giant Cobol and PL/SQL languages.

For a more elaborate discussion of metric values and their interpretation, we refer to [2].

3.4 Test suites

Functional 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^{++} [9]. We have constructed a functional test suite by collecting specifications from the internet[3]. 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 4 lists the suite of functional tests that we obtained in this way. Note that in spite of the small size of the functional 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 functional 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 3. As the chart indicates, all unit tests were developed during the disambiguation phase, i.e. between development versions 1 and 32.

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

Table 4. Functional test suite. The second column gives the number of code lines. The third and fourth columns gives coverage values for the final grammar.

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%

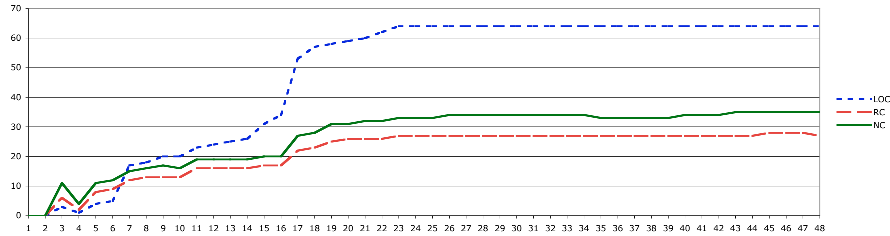

Fig. 3. The evolution of unit tests during development. The x-axis represents the 48 development versions. The three release versions among these are 1, 32, and 48. The left y-axis corresponds to lines of unit test code. Rule coverage (RC) and non-terminal coverage (NC) are shown as well.

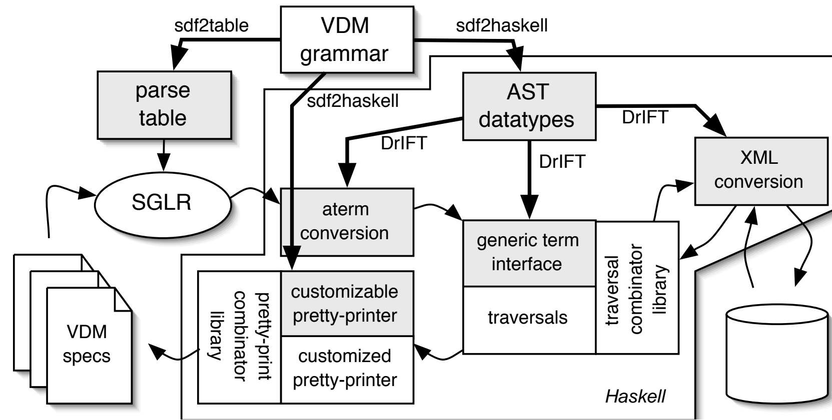

Fig. 4. Architecture of Strafunski. Bold arrows represent generators and grey boxes represent generated code.

4 VDM Tool Development

We have used the developed VDM grammar for construction of VDM tool support, following the grammar-centered approach outlined in Section 2.

4.1 Generation of support libraries

Given a high-quality, platform-independent grammar of VDM-SL, grammar-centered language tool development proceeds by generating components and libraries. For SDF grammars, such code generation is supported by various tools. To develop the VooDooM tool, we made use of the Haskell-based Strafunski bundle [19], of which the architecture is shown in Figure 4. This architecture is an instantiation of the general architecture of Figure 1, and we will briefly describe its main elements.

sdf2haskell From the VDM grammar, this tool generates (i) a Haskell module with datatypes that represent the VDM abstract syntax, and (ii) a Haskell module with a customizable pretty-printer that can convert ASTs into textual representation.

DrIFT [25] From the Haskell datatypes, this tool generates support libraries (i) for conversion between ASTs and the external ATerm representation, (ii) for providing ASTs with a generic term interface, and (iii) for conversion between ASTs and XML documents.

sdf2table [4] This generator produces a parse table from the VDM grammar, which can be consumed by the SGLR parser to parse VDM specifications into ATerms.

libraries A library of generic traversal combinators allows construction of traversals over the VDM ASTs using so-called strategic programming. A library of generic pretty-print combinators helps to customize the VDM pretty-printer.

user code The user uses the generated code and the libraries to construct a complete language processing tool, including a customized pretty-printer, and problem-specific AST traversals.

For targeting Java, similar support tools are available. In particular, the JJ-Forester tool generates Java class hierarchies and corresponding visitors from SDF grammars [17]. These generated class hierarchies and visitors enable generic and flexible traversal construction by instantiating the JJTraveler visitor combinator framework [24, 8].

4.2 Tool construction

In the case of VooDooM, the implemented traversals constitute a conversion engine that transforms sets of VDM datatypes into relational form. A second grammar, of SQL, was used to generate a pretty-printer for SQL, allowing export both to VDM and SQL. For a detailed explanation of the conversion engine and the use of strategic programming in its implementation, we refer to [1].

To give a brief indication of the style of programming that was employed, see the following code fragment for constructing the graph of dependencies between VDM types. This function is used in VooDooM to guide the transformations for recursion removal and inlining of types.

```
type Edge = (Identifier, Identifier)

dependGraph :: Document-> Gph Identifier
dependGraph d = mkRel (worker "" d)
 where
  worker :: Term a => Identifier -> a -> [Edge]
  worker parent d = maybe [] id (applyTU (stop_tdTU step) d)
   where
    step = failTU `adhocTU` def `adhocTU` use
```

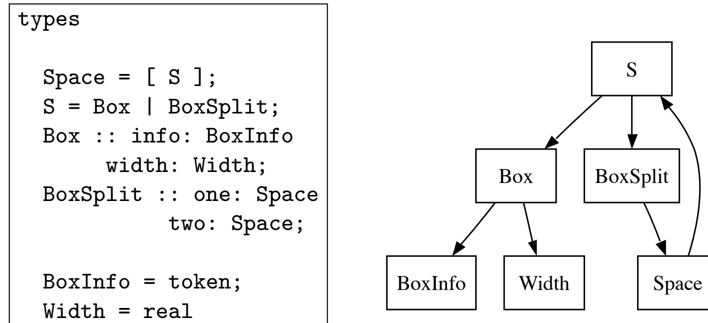

Fig. 5. A simple VDM document (left) and the corresponding type dependency graph (right) computed with the `dependGraph` function.

```
def :: TypeDefinition -> Maybe [Edge]
def (UnTaggedTypeDef id a b) = return $ worker id (a,b)
def (TaggedTypeDef id a b)   = return $ worker id (a,b)

use :: Type -> Maybe [Edge]
use (TypeName (Name id)) = return [(parent, id)]
use t = mzero
```

The `dependGraph` function invokes a `worker` function, which collects a list of edges, on its argument `d` and applies `mkRel` to construct a graph out of those edges. The worker takes an identifier that serves as parent node as additional argument, which is initialized with the dummy identifier `""`. The worker performs a top down collection strategy `stop_tdTU`, instantiated with a non-traversing `step` action. This `step` function in turn combines two helper functions for dealing with type definitions (`def`) and with used type names (`use`). The former restarts the worker with the defined identifier as new parent. The latter constructs an edge between the current parent and the identifier that names the used type. Note that this traversal works its way through all 71 Haskell datatypes, but only needs to mention a few of them explicitly. The input and output of the function for a simple VDM document is shown in Figure 5. A primer for traversal construction using generic traversal combinators is found in [18].

5 Concluding remarks

We have outlined a mix of grammar engineering techniques and discussed their application during the production of a VDM-SL grammar from its ISO specification. We have shown how such a grammar can be used in grammar-centered language tool development, involving generation of support libraries and generic traversal construction.

Availability The final version of the VDM-SL grammar in SDF (development version 48) is included in the appendix of [2]. 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/. The Haskell code generated from our VDM-SL grammars is distributed together with the grammar in the Vdm-Front package. The VooDooM tool is based on this generated front-end. Both of these can also be downloaded from the project web site.

Future work 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 are planning to provide generated Java support as well, by feeding our grammar to tools such as JJForester [17] or ApiGen [14]. The extensions would render our work compatible with the Java-based open source VDM tool development project, Overture[4]. It remains to be seen whether the generated artifacts would be integrated into previously developed components, or only used for new components.

Should the functionality of VooDooM be integrated into Overture, two possibilities exist. One is to regard the Haskell implementation as prototype for a new Java implementation. The other is to allow non-Java components to be plugged in to the Overture infrastructure as well. Especially for non-core functionality this latter option seems acceptable. If the same grammar is used to generate interchange support for Java and Haskell, seamless integration seems within reach.

Discussion A grammar-centered approach to language tool development has the advantage over more traditional approaches that it allows fast development of tools and components with different functionalities. We contend that formal method tools must offer more than a specification/programming language supporting traditional forward engineering. Rather, the potential benefits of a formally well-founded language such as VDM could be exploited to offer advanced functionality as exemplified by the VooDooM database model calculator.

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. Development of an industrial strength grammar for VDM. Technical Report DI-PURe-05.04.29, Universidade do Minho, April 2005.
3. T. Alves and J. Visser. Metrication of SDF grammars. Technical Report DI-PURe-05.05.01, Universidade do Minho, May 2005.
4. 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] See http://www.overturetool.org/

5. 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.
6. 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.
7. M. Bravenboer. Parse Unit home page. http://www.program-transformation.org/Tools/ParseUnit.
8. A. van Deursen and J. Visser. Source model analysis using the JJTraveler visitor combinator framework. *Softw. Pract. Exper.*, 34(14):1345–1379, 2004.
9. J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.
10. K.F. Fogel. *Open Source Development with CVS*. Coriolis Group Books, 1999.
11. M.H. Halstead. *Elements of Software Science*, volume 7 of *Operating, and Programming Systems Series*. Elsevier, New York, NY, 1977.
12. 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.
13. 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.
14. H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35–61, April-May 2004.
15. 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.
16. 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.
17. 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).
18. R. Lämmel and J. Visser. Design patterns for functional strategic programming. In *RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 1–14, New York, NY, USA, 2002. ACM Press.
19. 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.
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. J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, 2001. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001).
25. N. Winstanley. A type-sensitive preprocessor for haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, 1997. Most recent version available from http://repetae.net/john/computer/haskell/DrIFT/.