Universidade do Minho
Escola de Engenharia

Tiago Miguel Laureano Alves

VooDooM: Support for understanding and re-engineering of VDM-SL specifications

Tese de Mestrado
Mestrado em Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Nuno Oliveira Fonseca
Doutor Joost Visser

Maio de 2006

Preface

It is often the case that small decisions end up causing a big impact into one's life. It is funny look behind and see how everything started, all the path that I have been through, and all the people that accompanied me.

It all started in the 5th year Formal Methods course taught by Professor José Nuno Oliveira. My partner and friend Paulo Silva and I were challenged to create a simple prototype tool for database calculation from formal specifications (VooDooM). Although both of us were a bit sceptic at first, the project was successfully accomplished. A key factor for the project's success was the valuable help from Joost Visser, who introduced us to advanced software technologies such as "strategic term-rewriting" and "grammar engineering".

Meanwhile, in other 5th year project supervised by Joost Visser, Paulo, Bruno Herdeiro and I were asked to implement software analysis techniques for some mainstream programming languages in a bilateral project with the Dutch company Software Improvement Group (SIG). I still remember the first meeting, where all of us were sitting together discussing whether we were going to accept that work or not. I am glad we did, since this was one of the projects that changed me the most. It was the first time that I was faced with a completely different point of view to look at software: the *program understanding* point of view.

This project allowed us to caught a glimpse of the real-world software, by performing quantitative analysis of real-world systems in different languages. It was a tough challenge, but once more we succeeded. The involvement with SIG was profitable in other ways since they provided Paulo and me a PL/SQL

grammar that we used in the VooDooM project, thus saving a lot of work.

Both these works were not only professionally stimulating, but they have also captivated me with the charms of software science.

After finishing the first semester of the 5th year, I was faced with the choice of doing an internship at some company or doing it in the University. Influenced by professor José Nuno, and Dr. Luís Neves, I decided to embrace a new challenge, a University of Minho research project called IKF. The IKF project created all the conditions for starting a MSc, and once more professor José Nuno had a great impact in my decision by persuading me to accept. Once more I am glad that I followed his advice.

Many other persons had relevance to my work. One of them was professor Luís Barbosa who gently bought the ISO VDM Standard which allow me to build the tool front-end. Thank you for your trust.

With a hint from professor José Carlos Ramalho, after the IKF project I became teacher at University of Minho. This not only helped me to financially support myself, but also granted me some time to finish this thesis. Thus, I have to thank professors Pedro Henriques, João Saraiva and José João Almeida for all their support.

A special thanks to my supervisors, for all their patience and all the time and trust they invested in me.

Finally, I would like to thank all my friends for their support during all periods of this work, and for always remembering me what is important.

Thanks to Andreia for all the love and care, and for gently bringing me back down to earth every time my mind was somewhere else.

Thanks to my sisters and parents. Thanks mom, for all the hard work for me to have the best conditions to work and thanks dad, for the incentive by starting or ending all those long phone calls by asking "when are you going to finish you MSc?" or saying "You really have to finish your MSc!".

Thank you all, and it is to all of you that I dedicate this work.

VooDooM: Support for understanding and re-engineering of VDM-SL specifications

Abstract

The main purpose of this work is to define steady ground for supporting the understanding and re-engineering of VDM-SL specifications.

Understanding and re-engineering are justified by Lehman's laws of software evolution which state, for instance, that systems must be continually adapted and as a program evolves its complexity increases unless specific work is done to reduce it.

This thesis reports the implementation of understanding and re-enginering techniques in a tool called VooDooM, which was built in three well defined steps. First, development of the language front-end to recognize the VDM-SL language, using a grammar-centered approach, supported by the SDF formalism, in which a wide variety of components are automatically generated from a single grammar; Second, development of understanding support, in which graphs are extracted and derived and subsequently used as input to strongly-connected components, formal concept analysis and metrication. Last, development of re-engineering support, through the development of a relational calculator that transforms a formal specification into an equivalent model which can be translated to SQL.

In all steps of the work we thoroughly document the path from theory to practice and we conclude by reporting successful results obtained in two test cases.

VooDooM: Support for understanding and re-engineering of VDM-SL specifications

Resumo

O objectivo principal deste trabalho é a definição de uma infra-estrutura para suportar compreensão e re-engenharia de especificações escritas em VDM-SL.

compreensão e re-engenharia justificam-se pelas leis de evolução do software. Estas leis, formuladas por Lehman, definem, por exemplo, que um qualquer sistema deve ser continuamente adaptado e à medida que os programas evoluem a sua complexidade tende sempre a aumentar.

Esta tese descreve o estudo de técnicas de compreensão e re-engenharia que foram implementadas numa ferramenta chamada VooDooM. Esta implementação foi efectuada em três etapas bem definidas.

Primeiro, foi desenvolvido um parser (*front-end*) para reconhecer a linguagem VDM-SL. Para tal, foi utilizada uma abordagem centrada na gramática, suportada no formalismo SDF, que está equipado com ferramentas de geração automática de diversos componentes.

Segundo, para o suporte de compreensão, foram desenvolvidas funcionalidades para extrair e derivar grafos que são utilizados em técnicas de análise como componentes fortemente relacionados, análise de conceitos (formal concept analysis) e métricas.

Por último, para o suporte de re-engenharia, foi prototipada uma *calculadora relacional* que transforma um modelo, definido numa especificação formal, no seu equivalente relacional que pode ser traduzido para SQL.

Em todas as etapas realizadas h a preocupação de documentar o percurso entre teoria para a prática. A análise de resultados obtida no estudo de caso revela o sucesso da abordagem e as suas potencialidades para desenvolvimentos futuros.

Contents

Chapter 1

Introduction

The main purpose of this research is to define steady ground for supporting understanding and re-engineering of VDM-SL specifications.

Understanding will be accomplished through the extraction and derivation of different kinds of graphs which can be both visualized or subject of metrication. Re-engineering, on the other hand, will by achieved with the prototyping of a relational calculator which is able to convert a specification to its relational equivalent and hence, subsequently generate SQL.

The global contribution of this work is the VooDooM tool which supports understanding and re-engineering of VDM-SL specifications. Besides the general contribution, we specifically contribute with the following:

- A VDM-SL grammar which serves both as documentation and for generation of tool support,
- Powerful program understanding techniques that can be applied during development or to guide understanding,
- A database calculator which intends to promote database design through formal specification.

Our main motivation is the belief that formal methods and in particular VDM-SL lack proper tool support. Emphasis is put on development and validation but understanding and re-engineering is left to second plan. In this work we specifically intend to address those.

1.1 Motivation

Investigation about the evolution of OS/360 software led Lehman to formulate eight laws of Software Evolution [67].

Briefly, the first law, continuous change, states that a program that is used must be continually adapted else it becomes progressively less satisfactory; and the sixth law, continuing growth, states that functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

Moreover, Lehman states, that as a program evolves, its complexity increases unless work is done to maintain or reduce it (second law - increasing complexity); and programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment (seventh law - declining quality).

Formal specifications are normally used as a contract in which the most relevant system details are specified before they are implemented. When new requirements arise, they first are formally specified and after implemented. This describes an "one-way" only process, from specifications to the system's implementation.

Nevertheless, the contrary is also possible since it is feasible to do automatic re-engineering of parts of existing systems to formal specifications.

When formal specifications have such important role, they can be regarded as any other software artifact, being under "jurisdiction" of the same Lehman laws, and hence suffering the same evolution problems and requiring complexity and quality control.

The Vienna Development Method (VDM) [53, 32] is a collection of techniques for the formal specification and development of computing systems. VDM consists of a specification language, called VDM-SL, and an associated proof theory. Specifications written in VDM-SL describe mathematical models in terms of data structures and data operations that may change the model's state.

VDM-SL is quite a rich language in the sense that its syntax provides notation for a wide range of basic types, type constructors, and mathematical operators. VDM's origins lie in the research on formal semantics of programming languages at IBM's Vienna Laboratory in the 1960s and 70s, in particular of the semantics of PL/I. In 1996, VDM achieved ISO standardization [50].

A successful comercial tool supporting VDM-SL was developed by the Danish company IFAD. IFAD introduced extensions to the ISO VDM-SL [40] to allow executable specifications in order to make it a more appealing formal methods language. IFAD further extended the language with object oriented concepts calling it VDM++ [39]. VDM++ was supported by its own version of the VDMTools which additionally supported automatic code generation for C++ and Java, and re-engineering from VDM++ to Java [49].

ISO VDM-SL offers a clear notation and well-defined semantics. It is based on a mathematical model built from simple datatypes like sets, lists and mappings. Specifications can have state which is explicitly defined and manipulated by operations.

In this project we chosen to use ISO VDM-SL as model/specification language in program understanding and re-engineering, rather than using a set of ad-hoc or mathematically less rigorous notations. The motivation for this choice is that it allows us to develop techniques with a more solid theoretical foundation, and it brings into reach more sophisticated reasoning and calculation approaches to re-engineering and program understanding. Additionally, and in contrast with IFAD VDM-SL and VDM++, full semantics of the ISO VDM-SL[1] is available from [50].

Figure 1.1 provides an overview about how we interpret software re-engineering and understanding based on VDM-SL specifications.

[1]Whenever we mention VDM-SL or VDM, we will refer to ISO VDM-SL and not to any of their extensions.

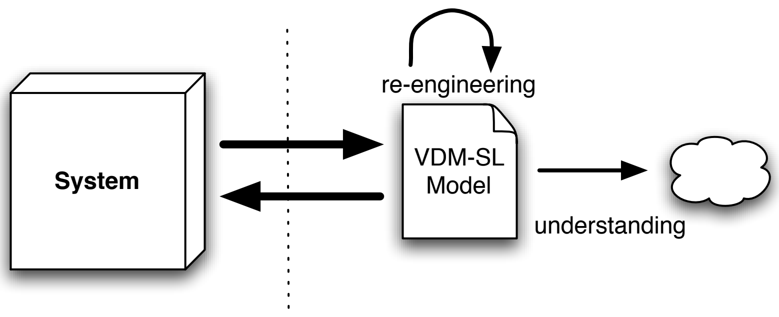


Figure 1.1: Re-engineering and understanding overview.

Our focus is the right side of the dashed lines, re-engineering and understanding of VDM-SL specifications, which can be obtained from a previous re-engineering project.

1.2 Program understanding

Associated with the system growth and complexity increase, Lehman's sixth and second law, respectively, there is also the problem of losing knowledge.

As a rule, lack of understanding is due not only to outdated documentation or unavailability of key personnel, but also to the sheer size and complexity to which systems grow over time. When this happens, design decisions, system's architecture and, sometimes, program semantics become entangled in the extensive details of the source code.

Program understanding [85] is the process of acquiring knowledge about programs and provides valuable help in solving this problem. Program understanding can achieve this by using reverse engineering techniques based on source code analysis.

Many program understanding techniques used today were borrowed from other areas. Data and control flow analysis were first used in optimizing compilers [3]. Data flow analysis is used for finding dependencies between different data items manipulated by a program and it can be used to discover relationships in data, verify properties in concurrent programs, and others. Control flow analysis [77] allows for execution of a given program to be discovered, and to identify dependencies in the execution of code. These two, combined with *slicing* [90, 77], for isolating specific parts of the program code, are powerful techniques [77] used for automatic debugging, property checking and validation, and aiding program understanding.

Architecture recovery has been subject of research in order to assist migration of legacy systems but also to validate systems design. Techniques such as cluster and concept analysis have been developed and applied with some success [93].

Quantitative analysis, or *software metrics*, is used to assign numeric values to particular aspects of software. Halstead, McCabe and Allbrecht in the late seventies are credited for defining the most well-known metrics today. The *Halstead metrics* [41] were developed to measure program module complexity from the operators and operands they use. McCabe defined *cyclomatic complexity* [71] as the number of linearly-independent paths through a program module. Allbrecht defined *function point analysis* [4] as a means of measuring software size and productivity using functional, logical entities such as inputs and outputs. After almost two decades, in a effort to measure and track maintainability, the Maintainability Index [104] was proposed, combining Halstead, McCabe and other simple metrics.

1.3 Program re-engineering

Program re-engineering can be defined as “examination and alteration of a system to rebuild it in a new form and subsequent implementation of the new form” [20].

Different software engineering processes fit in this category, namely software maintenance and software renovation.

Forward engineering [21], also called reification or refinement in formal methods, is the process of deriving a lower level model, specification, or program from a higher-level one. Thus, to reify (or refine) is to make it into ‘a more concrete thing’.

Refinement can also be divided into two categories, data and operation refinement. Converting from an abstract data model type such as a product type, into its implementable data structure such as a C *struct* or a Java *class*, are examples of data refinement. Examples of operation refinement are, for instance, the conversion of a specification of an operation into an implementable program such as a C procedure or Java method.

Because formal specification languages have a strong mathematical basis, these transformations can be methodically derived and mathematical proofs can be done to prove their correctness.

In VDMTools, the refinement process is implemented targeting C++ or Java.

In [26], the author investigated the refinement process between the two dialects of VDM, by converting functional specifications written in VDM-SL to VDM++. In this work, we are interested in exploring the possibility of targeting other models, specifically the SQL model [34].

1.4 Objectives

The general objective of the work is to provide basic support for understanding and re-engineering. This requires development of a language front-end, or parser, for VDM-SL and development of understanding and re-engineering functionalities.

For each of these items mode detailled objectives will be given next:

VDM-SL front-end Develop an SDF grammar using strong software engineering practices applied to grammar development such as grammar versioning, metrication, visualization, and testing.

Program understanding Implement a few static analysis techniques like type dependency, function call and function and type dependency graphs.

The results of these analysis will be fed to both visualization and quantitative analysis. The novelty that will be introduced here, is the derivation of metrics (quantification analysis) which we expect to provide us insights over the complexity or structure of systems formal specifications.

Program re-engineering Introduce a novel approach for database design through its specification using a VDM-SL model. The more abstract model specified in VDM-SL will be refined to an SQL database schema. We will provide not only a detailed overview over the method that can support database specification, but also about the transformation process.

Throughout this work, we show how these techniques can be elegantly and concisely implemented in the Haskell [54, 89] programming language using strategic term rewriting provided by the Strafunski [66] bundle.

As proof of concept, a tool called VooDooM has been developed.

1.5 General approach

In order to achieve the objectives just stated, a roadmap was set up (Figure 1.2) which reflects both the architecture of the intended, and the thesis structure. This is as follows:

First of all, background about the technology adopted to accomplish the whole work will be presented in Chapter 2.

The first step of the work was the implementation of a language recognition component: a parser for the VDM language (represented at the bottom of Figure 1.2). The development of this component is of great importance since all of the work relies on this component. This requires strong grammar engineering techniques in order to achieve the required high-quality. Both the engineering techniques and the development method will be fully described in Chapter 3.

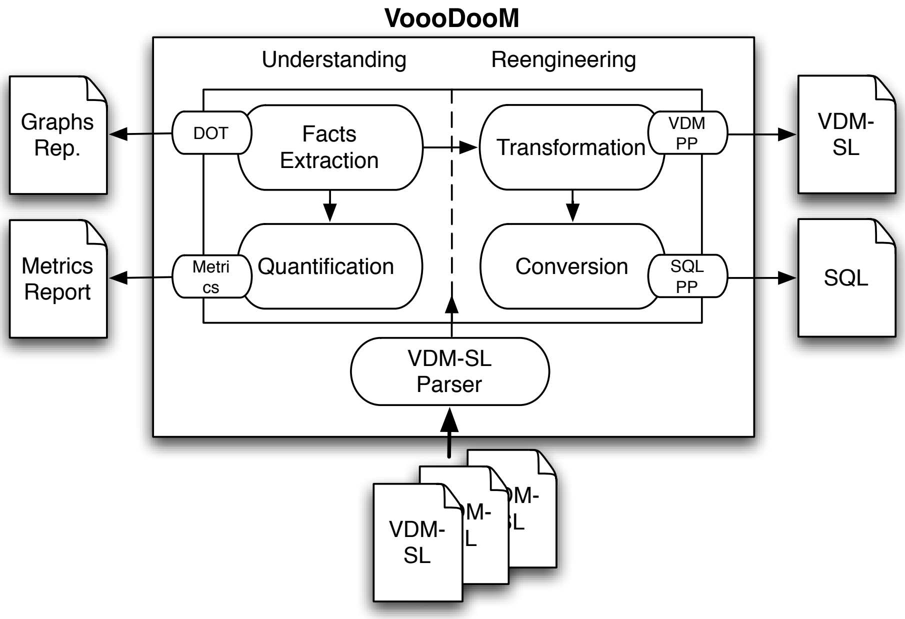


Figure 1.2: Overall architecture of the VooDooM *tool.*

For program understanding, represented on the left hand side of Figure 1.2, two main components, "Facts extraction" and "Quantification", have been developed. The "Facts extraction" module is responsible for extracting information such as procedure calls, type dependencies and procedure-type flow. The results of this module can be output using the "DOT" module, for visualization purposes, or fed to the "Quantification" module. The "Quantification" module is responsible for metrics calculation and its result is output as a metrics report.

All the work done for program understanding is detailed in Chapter 4. The program re-engineering part, represented on the right hand side of Figure 1.2, supports database calculation in two steps, represented by the "Transformation" and "Conversion" components. The "Transformation" component implements the refinement process by transforming the input model into an equivalent VDM model. The result can be both output to the "VDM-SL PP" component, that prints it as a VDM specification (through "VDM-SL PP" component) and to the “Conversion” component that converts it to SQL and prints it (through the “SQL PP” component). Chapter 4 documents both the method and the implementation of database calculation.

Finally, the conclusions and future work are presented in Chapter 6.

1.6 Origins of the chapters

Sections 2.1, 3, 6.2.1, 6.3.1, and 6.4.1 were based on the work co-authored by Joost Visser, submitted for publication:

> T. Alves and J. Visser. Grammar engineering applied for development of a VDM grammar. [7]

Sections 2.3, 5, 6.2.3, 6.3.3, and 6.4.3 were based on the work co-authored by Paulo Silva, Joost Visser and José Nuno Oliveira, published earlier as:

> T.L. Alves, P.F. Silva, J. Visser, and J.N. Oliveira. Strategic term rewriting and its application to a VDM-SL to SQL conversion. In *Proceedings of FM 2005: 13th International Symposium on Formal Methods*, In Lecture Notes in Computer Science 3582, pages 399-414. Springer, 2005. [6]

Chapter 2

Technology

The aim of this chapter is to introduce basic notions about the technology adopted, as follows:

Section 2.1 introduces *grammar-centered tool development*, which advocates the generation of several components from a single grammar. These components, as we will explain, are the foundation of the language front-end.

Section 2.2 introduces SDF, the formalism in which grammars are specified. We will also provide a brief overview of the available tools.

Finally, Section 2.3 introduces a library of combinators which add support for generic traversal in Haskell [54, 89]. This library, called StrategyLib available in the Strafunski [66] bundle developed by Ralf Lämmel and Joost Visser, is extensively used in the implementation of both analysis and transformation rules.

2.1 Grammar-centered tool development

One of the first challenges for language tool support is the development of a language front-end (parser). In traditional approaches, the grammar of the language is encoded in a parser specification. Commonly used parser generators include Yacc/Bison [51], Antlr [82], and JavaCC [58]. However, the parser specifications read by such tools are not general context-free grammars.

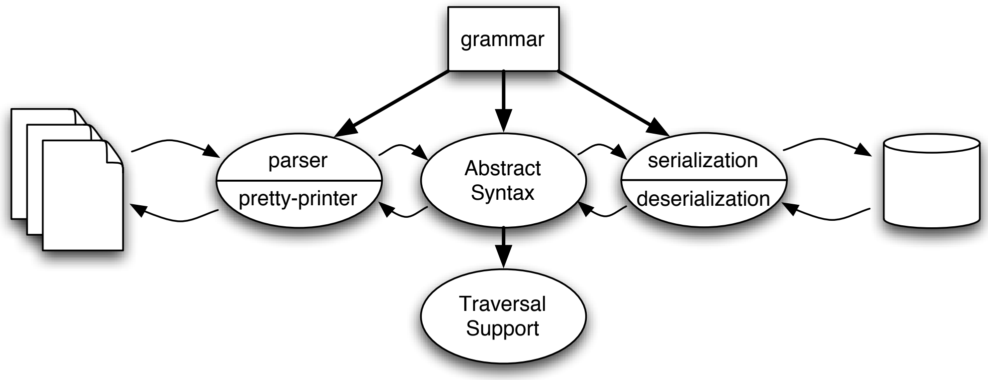


Figure 2.1: Grammar-centered approach to language tool development.

Rather, they are grammars within a proper subset of the class of context-free grammars, such as LL(1), or LALR [3]. 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: to generate a parser in a single programming language, with a single 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 [92].

For language tool support we will use a *grammar-centered* approach [28]. 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 de-serialization in various formats, customizable pretty-printers, and support for syntax tree traversal. This approach is illustrated by the diagram in Figure 2.1.

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,

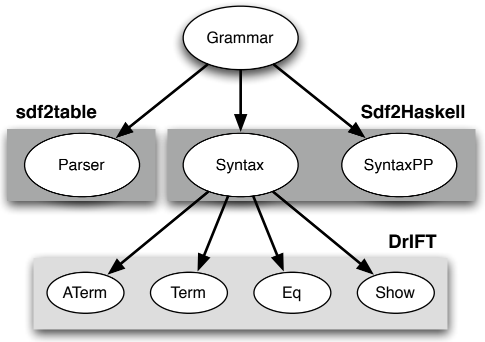


Figure 2.2: Grammar-centric approach diagram.

and it should leave the specification of semantics to a specific layer, but be concerned with syntax only. Since grammar changes potentially lead to changes in many related tools, 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.

Possible candidates are BNF or EBNF, or our grammar description of choice: SDF [43, 94]. The reason for this choice is that SDF is the only formalism that has tool support and it offers an excellent integration with Haskell. SDF will be introduced in Section 2.2.

The grammar centered approach is supported in the Haskell programming language via tools such as eg. *sdf2table*, *Sdf2Haskell* and *DrIFT* as illustrated in Figure 2.2.

Parse tables are automatically generated by the *sdf2table* tool from the SDF software bundle, which corresponds to the *Parser* ellipses of Figure 2.2.

The AST Haskell datatype definition and the pretty-printer are generated with the *Sdf2Haskell* tool from Strafunski [66]. They correspond to the *Syntax* and *SyntaxPP* ellipsis in the picture.

From the abstract syntax, further components are generated in the form of Haskell class instances, using the *DrIFT* tool. The *ATerm* instances support serialization to the *ATerm* format, which is used as interchange

```
context-free syntax

   "mk_" Name "(" ExpressionList? ")" -> Expression { cons("RecordConstructor") }

   "let" { LocalDefinition "," }+ "in" Expression -> Expression { cons("LetExpr") }
```

Figure 2.3: Excerpt of the Expressions module of the VDM-SL grammar.

format between the generated parser and other components. The *Term* instances support generic traversal and strategic term rewriting over ASTs. The last two (*Eq* and *Show*), are not mandatory: they add comparison and printing functions to the Haskell datatypes.

2.2 SDF by example

The syntax definition formalism [94], or SDF for short is a simple notation to define grammars. Figure 2.3 presents an example of SDF notation (an excerpt of the VDM-SL grammar [50]).

One of the most noticeable differences between (E)BNF and SDF is that SDF rules are written in reverse order. Moreover, SDF allows for the specification of even more regular expression-style constructs to BNF than EBNF does, such as separated lists, for instance.

In the example of Figure 2.3, non-terminal `Expression` is defined with two rules. The first rule specifies `Expression` as being a terminal "`mk_`", followed by a `Name`, and an optional `ExpressionList` between brackets. The second rule specifies `Expression` as being a terminal "`let`", a list with one or more elements of `LocalDefinition` separated by a comma, a terminal "`in`", and an `Expression`.

Alternatives can either be specified as (E)BNF with the | operator (as in Yacc [51] or BNF) or just by adding a new rule defining the same non-terminal (as shown in the example).

Yacc or its sucessor Bison allows grammar rules[1] to have semantic actions

[1]Actually semantic actions can be specified just after any symbol in a rule and are

```
module VDM-Syntax
   imports Characters Layout
   exports sorts SymbolicLiteral Identifier QuoteLiteral

   context-free syntax
      "true"  -> BooleanLiteral { cons("TRUE") }
      "false" -> BooleanLiteral { cons("FALSE") }
      ...

   lexical syntax

      Numeral ( "." Digit+)? Exponent? -> NumericLiteral
      Digit+ -> Numeral
      "E" ("+" | "-")? Numeral -> Exponent
```

Figure 2.4: An excerpt of module `VDM-Syntax`*.*

made up of statements in the C [56] programming language.

Although this enables doing some processing during parsing, this implies that the grammar can be only used for creating a parser generator in a single programming language (in this particular case, C). SDF overcomes this by being purely declarative, i.e., not allowing for semantic actions. However, SDF allows for the specification of rule attributes such as, for instance, disambiguity and abstract syntax tree constructs. In the example of Figure 2.3 the latter kind is shown.

Additionally, in contrast to (E)BNF, SDF has the important advantage of allowing one to specify both context-free and lexical syntax and it offers a flexible modularization mechanism. The module mechanism accepts mutually dependent modules, and distribution of alternatives of the same non-terminal across multiple modules.

In Figure 2.4 an excerpt of the module `VDM-Syntax` is shown in which the module import and export mechanisms, the context-free section and the lexical syntax are used. Besides, SDF also allows the user to specify restrictions and priorities sections which provide a mechanism for disambiguation.

SDF is supported by several tools such as a well-formedness checker, a GLR

executed if that symbol is matched.

parser generator, generators of abstract syntax support for various programming languages (among which Java [9], Haskell [54, 89], and Stratego [95]), and customizable pretty-printer generators [16, 95, 66, 59, 55].

2.3 Strategic term rewriting

Traditional term rewriting distinguishes the rewriting *equations* of a particular term rewriting system (TRS) from the *strategy* that is used to apply these equations to an input term. Most commonly, term rewriting environments has a fixed rewriting strategy, such as the leftmost-innermost strategy. In some rewriting environments, for instance those where the equations may be governed by conditions and may be stratified into default and regular equations, more sophisticated strategies may be employed. In any case, however, these strategies are fixed, i.e. hard-wired into the environment.

By contrast, strategic term rewriting generalizes the traditional term rewriting paradigm by making rewriting strategies programmable, just as the equations are. Stratego [95] and the Rewriting Calculus [22] are among the first rewriting environments to offer such programmable rewriting strategies. Such environments offer a small set of basic strategy combinators, which can be combined with each other and with rewriting equations to construct term rewriting systems with arbitrarily complex strategies.

Figure 2.5 shows a set of such basic strategy combinators, along with their operational semantics, from which more elaborate ones can easily be constructed. Consider for instance the following definitions:

$$\begin{aligned}
&\mathit{try}(s) &&= \mathsf{choice}(s, \mathsf{id})\\
&\mathit{repeat}(s) &&= \mathit{try}(\mathsf{seq}(s, \mathit{repeat}(s)))\\
&\mathit{full\_topdown}(s) &&= \mathsf{seq}(s, \mathsf{all}(\mathit{full\_topdown}(s)))\\
&\mathit{innermost}(s) &&= \mathsf{seq}(\mathsf{all}(\mathit{innermost}(s)), \mathit{try}(\mathsf{seq}(s, \mathit{innermost}(s))))
\end{aligned}$$

The *try* combinator takes a potentially failing strategy as argument, and attempts to apply it. Should failure occur, the identity strategy id is used to recover. The *repeat* combinator repeatedly applies its argument strategy, until

Combinators

s	$::=$	id	Identity strategy
	$\mid$	fail	Failure strategy
	$\mid$	$\mathsf{seq}(s, s)$	Sequential composition
	$\mid$	$\mathsf{choice}(s, s)$	Left-biased choice
	$\mid$	$\mathsf{all}(s)$	All immediate components
	$\mid$	$\mathsf{one}(s)$	One immediate component
	$\mid$	$\mathsf{adhoc}(s, a)$	Type-based dispatch

Notation

d	...	data
c	...	data constructors
$\overline{d}$	...	data with failure "$\uparrow$"
a	...	type-specific actions
s	...	strategies
$a@d$	...	application of a to d
$s@d$	...	application of s to d
$d \Rightarrow \overline{d}$	...	big-step semantics
$a : t$	...	type handled by a
$d : t$	...	type of a datum d
$[d]$	...	indivisible data
$c(d_1 \cdots d_n)$	...	compound data

Meaning

$\mathsf{id}@d$	$\Rightarrow$	d
$\mathsf{fail}@d$	$\Rightarrow$	$\uparrow$
$\mathsf{seq}(s, s')@d$	$\Rightarrow$	$\overline{d}$ if $s@d \Rightarrow d' \wedge s'@d' \Rightarrow \overline{d}$
$\mathsf{seq}(s, s')@d$	$\Rightarrow$	$\uparrow$ if $s@d \Rightarrow \uparrow$
$\mathsf{choice}(s_1, s_2)@d$	$\Rightarrow$	d' if $s_1@d \Rightarrow d'$
$\mathsf{choice}(s_1, s_2)@d$	$\Rightarrow$	$\overline{d}$ if $s_1@d \Rightarrow \uparrow \wedge s_2@d \Rightarrow \overline{d}$
$\mathsf{all}(s)@[d]$	$\Rightarrow$	$[d]$
$\mathsf{all}(s)@c(d_1 \cdots d_n)$	$\Rightarrow$	$c(d_1' \cdots d_n')$ if $s@d_1 \Rightarrow d_1', \ldots, s@d_n \Rightarrow d_n'$
$\mathsf{all}(s)@c(d_1 \cdots d_n)$	$\Rightarrow$	$\uparrow$ if $\exists \cdot i. \overleftarrow{s}@d_i \Rightarrow \uparrow$
$\mathsf{one}(s)@[d]$	$\Rightarrow$	$\uparrow$
$\mathsf{one}(s)@c(d_1 \cdots d_n)$	$\Rightarrow$	$c(\cdots d_i' \cdots)$ if $\exists \cdot i. \overleftarrow{s}@d_1 \Rightarrow \uparrow \wedge \cdots \wedge s@d_{i\text{-}1} \Rightarrow \uparrow \wedge s@d_i \Rightarrow d_i'$
$\mathsf{one}(s)@c(d_1 \cdots d_n)$	$\Rightarrow$	$\uparrow$ if $s@d_1 \Rightarrow \uparrow, \ldots, s@d_n \Rightarrow \uparrow$
$\mathsf{adhoc}(s, a)@d$	$\Rightarrow$	$a@d$ if $a : t$ and $d : t$
$\mathsf{adhoc}(s, a)@d$	$\Rightarrow$	$s@d$ if $a : t \wedge d : t' \wedge t \neq t'$

Identities

[unit]	s	$\equiv$	$\mathsf{seq}(\mathsf{id}, s)$	$\equiv$	$\mathsf{seq}(s, \mathsf{id})$	$\equiv$	$\mathsf{choice}(\mathsf{fail}, s)$	$\equiv$	$\mathsf{choice}(s, \mathsf{fail})$
[zero]	fail	$\equiv$	$\mathsf{seq}(\mathsf{fail}, s)$	$\equiv$	$\mathsf{seq}(s, \mathsf{fail})$	$\equiv$	$\mathsf{one}(\mathsf{fail})$		
[skip]	id	$\equiv$	$\mathsf{choice}(\mathsf{id}, s)$	$\equiv$	$\mathsf{all}(\mathsf{id})$				

[nested type dispatch]

$\mathsf{adhoc}(\mathsf{adhoc}(s, a), a')$	$\equiv$	$\mathsf{adhoc}(s, a')$ if $a : t \wedge a' : t$
$\mathsf{adhoc}(\mathsf{adhoc}(s, a), a')$	$\equiv$	$\mathsf{adhoc}(\mathsf{adhoc}(s, a'), a)$ if $a : t \wedge a' : t' \wedge t \neq t'$
$\mathsf{adhoc}(\mathsf{adhoc}(\mathsf{fail}, a), a')$	$\equiv$	$\mathsf{choice}(\mathsf{adhoc}(\mathsf{fail}, a), \mathsf{adhoc}(\mathsf{fail}, a'))$ if $a : t \wedge a' : t' \wedge t \neq t'$

Figure 2.5: Specification of a guideline set of basic strategy combinators.

it fails. The *full_topdown* combinator applies its argument once to every node in a term, in pre-order. Finally, the *innermost* strategy applies its argument in left-most innermost fashion to a term, until it is not applicable anywhere anymore, i.e. until a fixpoint is reached.

For a more in-depth explanation of these combinators we refer the reader to [96, 65].

The challenge of combining strategic term rewriting with strong typing was first met by the Haskell-based Strafunski bundle [66], which we will use in this thesis, and the Java-based JJTraveler framework [96, 59]. A formal semantics of typed strategic programming is defined in [61]. Further generalizations were provided in the Haskell context [65, 63].

Strategic term rewriting has several benefits over traditional term rewriting. The most important benefits derive from the fact that many applications require rewrite equations that together do not form a confluent and terminating TRS. A program refactoring system, for instance, may require equations both for "extract method" and for "inline method". A document processing system may include equations that change mark-up only inside the context of certain document tags. In a traditional term rewriting environment, the only option to obtain sufficient control over when and where equations are applied, is to switch to so-called 'functional style'. This means that every rewrite rule $t \mapsto \ldots s \ldots$ is reformulated to include function symbols to control rewriting: $f(t) \mapsto \ldots g(s) \ldots$. In this way, the rewriting strategy becomes explicit in the additional function symbols, but is thoroughly entangled with the rewrite equations. In strategic programming, the rewrite equations can stay as they are, the strategy is specified separately, and both equations and strategies can be used and reused in different combinations to obtain different TRSs. So, apart from full control over when and where equations are applied, strategic rewriting enhances separation of concerns, reusability, and understandability.

In this work, we will rely on strategic term rewriting to cleanly separate the individual rules from the strategy of applying them to the abstract syntax terms. The Strafunski bundle is our choice of strategic term rewriting environment, in which we will implement a understanding and re-engineering tool called VooDooM.

Chapter 3

A front-end for VDM-SL

In this chapter, we provide details about the grammar engineering part of this project, in which a full grammar of the VDM-SL [50] language is obtained from its ISO standard language reference. We do so with two complementary objectives.

Firstly, we intend to convey our experiments with the application of a mix of grammar engineering techniques, thus contributing to the body of knowledge about best grammar engineering practises. Our approach can be characterised as a tool-based methodology for iterative grammar development embedded into the larger context of grammar-centered language tool development.

Secondly, we aim to document the delivered grammar of VDM-SL, as well as the steps that led to its creation and account for its quality. A well-engineered grammar can significantly reduce the risks and effort involved in subsequent language tool development, and developers of VDM processing tools need to be convinced of the quality of the grammar on which their product hinges.

These two objectives are mutually complementary, in the sense that the application of the techniques validate their practical value and scalability, and that the quality of the grammar is strongly dependent on the techniques used for its development. In accordance with this two-fold aim, this chapter is structured as follows.

Section 3.1 describes our tool-based methodology for grammar engineering, independent of any specific grammar development project. We address key issues such as grammar versioning, metrication, visualization, and testing, and we embed our grammar engineering approach into the larger context of grammar-centered language tool development. Section 3.2 describes the application of these general techniques in the specific case of our VDM grammar development project. We describe the development process along with its intermediate and final deliverables.

3.1 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.

In this section we will discuss the grammar engineering techniques that we adopted, and how we adapted them to the specific process of developing SDF grammars.

3.1.1 Grammar evolution

Grammars for sizeable languages are not created in a single goal: they arise through prolonged, resource consuming processes. 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 [33].

In grammar evolution, different kinds of transformation steps occur:

Recovery: An initial version of the grammar may be retrieved by reverse engineering from an existing parser, or by converting a language reference manual, available typically as a Word or PDF document. If only a hardcopy is available then it should be typed in.

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 (sub) language.

Refactoring: Changing the shape of the grammar, without changing the language that it generates. 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.

3.1.2 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 [83] and implemented a tool, called SdfMetz citeDI-PURe-05.05.01, to collect grammar metrics for SDF grammars.

Size and complexity metrics	
TERM	Number of terminals
VAR	Number of non-terminals
MCC	McCabe's cyclometric complexity
AVS-P	Average size of RHS per production
AVS-N	Average size of RHS per non-terminal

Figure 3.1: Size and complexity metrics for grammars.

Some adaptation was necessary because 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, in order to make comparisons possible with the results of others. For the disambiguation information these metrics were extended with the definition of a dedicated set of metrics. Full details about the definition and the implementation of these SDF metrics are provided in [8].

We will discuss several categories of metrics: size and complexity metrics, structure metrics, Halstead metrics, and disambiguation metrics by providing a brief description of each.

Size, complexity, and structure metrics

Figure 3.1 lists a number of size and complexity metrics for grammars. These metrics are defined for BNF in [83]. The number of terminals (TERM) and non-terminals (VAR) are simple metrics applicable both to BNF and SDF grammars. 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.

Structure metrics	
TIMP	Tree impurity (%)
CLEV	Normalized count of levels (%)
NSLEV	Number of non-singleton levels
DEP	Size of largest level
HEI	Maximum height

Figure 3.2: Structure metrics for grammars.

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, and divide 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). For grammars where each non-terminal has a single production rule, as is the case for (E)BNF grammars, these metrics will present the same value. For SDF grammars, the values can be different. While the AVS-N metric is more appropriate to compare with other formalisms (like BNF and EBNF), the AVS-P metric provides more precision.

Structure metrics

Figure 3.2 lists a number of structure metrics also previously defined in [83]. Each of these metrics is based on the representation of a grammar as a graph which has non-terminals as nodes, and which contains edges between two non-terminals whenever one occurs in the right-hand side of the definition of the other. This graph is called the grammar's flow graph.

Tree impurity (TIMP) measures how much the flow 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 a fully connected graph.

Only the tree impurity metric (TIMP) is calculated directly from the flow graph, all the other structure metrics are calculated from the corresponding strongly connected components graph. This latter graph is obtained from the flow graph by grouping the non-terminals that are strongly connected (reachable from each other) into nodes (called components of levels of the grammar). An edge is created from one component to another if in the flow graph at least one non-terminal from one component has an edge to a non-terminal of the other component. This graphs is called the grammar's level graph or graph of strongly connected components.

Normalized count of levels (CLEV) expresses the number of nodes in the level graph (graph of strongly connected components) as a percentage of the number of nodes in the flow graph. A normalized count of levels of 100 percent means that there are as many levels in the level graph as non-terminals in the flow graph. In other words, there are no circular connections in the flow graph, and the level graph only contains singleton components. A normalized count of levels of 50 percent means that about half of the non-terminals of the flow graph are involved in circularities and are grouped into non-singleton components in the level graph.

Number of non-singleton levels (NSLEV) indicates how many of the grammar levels (or strongly connected components) contain more than a single non-terminal.

Size of the largest level (DEP) measures the depth (or width) of the level graph as the maximum number of non-terminals per level.

Maximum height (HEI) measures the height of the level graph as the longest vertical path through the level graph, i.e. the largest path length from a source of the level graph to a sink.

Halstead metrics

The Halstead Effort metric [41] has also been adapted for (E)BNF grammars in [83]. We compute values not only for Halstead's effort metric but also

Halstead metrics	
n1	Number of distinct operators
n2	Number of distinct operands
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 (HAL)
L	Program level
T	Program time

Figure 3.3: Halstead metrics for grammars.

for some of its ingredient metrics and related metrics. Figure 3.3 shows the 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 extend this interpretation from BNF to SDF we refer the reader again to [8].

The theory of software science behind Halstead's metrics has been widely questioned. In particular, the meaningfulness and validity of the effort and time metrics have been called into question [30]. Below, we will still report HAL-E, for purposes of comparison to data reported in [83].

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 define a series of metrics, which are shown in Figure 3.4. These metrics are simple counters for each type of ambiguity construct offered by the SDF notation.

Ambiguity metrics	
FRST	Number of follow restrictions
ASSOC	Number of associativity attributes
REJP	Number of reject productions
UPP	Number of unique productions in priorities

Figure 3.4: Ambiguity metrics for grammars.

3.1.3 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 behavioral 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 files as test data. The grammar is tested by generating a parser from it and running this parser on such files. Test observations are the success or failure of parsing input files, possibly including time and space consumption. A unit test will use fragments of files as test data. Typically, such fragments are composed by grammar developers to help in detecting and solving specific errors in the grammar, and to protect themselves 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.

For both functional and unit testing we have used the `parse-unit` utility [18]. Tests are specified in a simple unit test description language with which it is possible to declare whether a certain input should parse or not, or that a certain input sentence should produce a specific parse tree (tree shape testing). Taking such test descriptions as input, the `parse-unit` utility allows batches of unit tests to be run automatically and repeatedly.

3.1.4 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 tool to determine grammar test quality is coverage analysis. We have adopted the rule coverage (RC) metric [84] 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.

SDF allows for two possible interpretations of RC, due to the fact that a single non-terminal may be defined by multiple productions. (Above, we discussed a similar interpretation problem for the AVS metric.) One way is to count each of these alternative productions separately. Another way is to count different productions of the same non-terminal as one. For comparison with rule coverage for (E)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. These numbers were computed for our functional test suite and unit test suite by a tool developed for this purpose, called SdfCoverage [8].

An even more accurate indication can be obtained with context-dependent rule coverage [60]. This metric takes into account not just whether a given production is used, but also whether it has been used in every context (use site) where it can actually occur. However, implementation and computation of this metric is more involved.

3.2 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 describe the scope, priorities, and planned deliverables of the project, as well as its execution. We refer to the evolution of the grammar during its development both in qualitative and quantitative terms, using the metrics described above. The test effort during the project is described in terms of the test suites used and the evolution of the unit tests and test coverage metrics during development.

3.2.1 Scope, priorities, and planned deliverables

Language Though we are interested in eventually developing grammars for various existing VDM dialects, such as IFAD VDM and VDM++ [40, 39], we limited the scope of the initial project to the VDM-SL language as described in the ISO VDM-SL standard [50].

Grammar shape Not only should the parser generate the VDM-SL language exactly as defined in the standard, but also the shape of the grammar, the names of the non-terminals, and the module structure should correspond closely to the grammar. Moreover, 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 developing a grammar from which a GLR parser cold 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 closely as possible to the abstract syntax as defined in the ISO standard.

Planned deliverables 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 SDF's regular expression-style 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. The grammar's shape should be changed to better reflect the tree shape as intended by the abstract syntax in the standard.

3.2.2 Grammar creation and evolution

To accurately keep track of all grammar changes, a new revision was created for each grammar evolution step. 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 initial version of the grammar was typed in from the hardcopy of the ISO Standard [50]. In this document, context-free syntax, lexical syntax and disambiguation information are specified in a semi-formal notation. Context-free syntax is specified in EBNF, but the terminals are specified as mathematical symbols. Translating these mathematical symbols to ASCII symbols involves an interchange table. Lexical syntax is specified in tables by enumerating all possible symbols. Finally, disambiguation information is specified in terms of precedence in tables and equations.

Apart from changing syntax from EBNF to SDF and using the interchange table to replace mathematical symbols by their parseable representation, the following actions were involved in the transcription.

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

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

During transcription this was converted to:

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

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

Detected top and bottom non-terminals To help in the manual process of typing the grammar, a small tool was developed to detect top and bottom non-terminals. This tool helped to detect typos. More than one top non-terminal, or a bottom non-terminal 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 EBNF does not support modularization. The ISO Standard separates concerns by dividing the EBNF rules over sections. SDF does support modules, which allowed us to modularize the grammar following the sectioning of the ISO standard. Moreover, another small tool was implemented to discover the dependencies between modules.

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 ad hoc 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 additional definitions. Detection of top and bottom non-terminals in this case helped to detect some inconsistencies in the standard such as name consistency.

Disambiguation

In SDF, disambiguation is specified by means of dedicated disambiguation constructs [17]. 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 to 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 that follow the ISO standard as close as possible. Two operations were performed:

- adding constructor attributes to the contex-free rules to specify AST node labels
- removing injections to make the grammar and the ASTs nicer to read

How the AST will look with the construct attributes depends to the target language. An example of a construct attribute and the result in Haskell can be read as follows:

Version	TERM	VAR	MCC	AVS-N	AVS-P	HAL-E	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 3.1: Grammar metrics for the three release versions.

```
Type "*" Type -> Type { left, const("ProducType") }
...


data Type = ProductType Type Type
          | ...
```

Note that the construct attribute name was used as selector from the type, while the name of type of the rule was used as the Haskell type.

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, which already exist in the original EBNF grammar, were actively removed, because they needlessly increase the size of the grammar (which can be observed in the measurements) and degrade readability. Also, the corresponding automatically built ASTs are more compact after injection removal.

3.2.3 Grammar metrics

We measured grammar evolution in terms of the size, complexity, structure, and Halstead metrics introduced above. The data is summarized in Table 3.1. This table shows the values of all metrics for the three released versions. In addition, Figure 3.5 plots the evolution of a selection of these metrics for all 48 development versions.

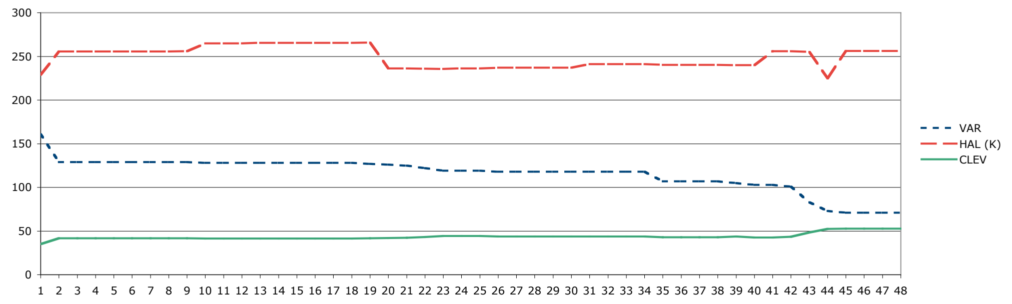


Figure 3.5: The evolution of VAR, HAL-E, and CLEV grammar metrics during development. The x-axis represents the 48 development versions.

Size and complexity metrics

A first important observation to make is that the number of terminals (TERM) 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 (VAR) 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. The reason for this refactoring during the disambiguation phase was to make disambiguation easier. 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 using priorities became possible.

The average rule 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 size of 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.

Halstead metrics

The value of the Halstead Effort metric (HAL-E) 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 quietly 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. Below, a comparison of these values with those of other grammars will be offered.

Structure metrics

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, or, in other words, that complexity due to circularities 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 inlining 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 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 tells 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 to inlining.

The size of the largest grammar 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, which corresponds to *Expressions*. The decrease in level sizes is directly attributable to inlining of grammar rules involved in cycles.

The height of the level graph (HEI) is 16 throughout most of the evolution of the grammar, but sinks slightly to 14 towards the end of development. Only inlining of production rules not involved in cycles leads to reduction of path length through the level graph. This explains why the decrease of HEI is modest.

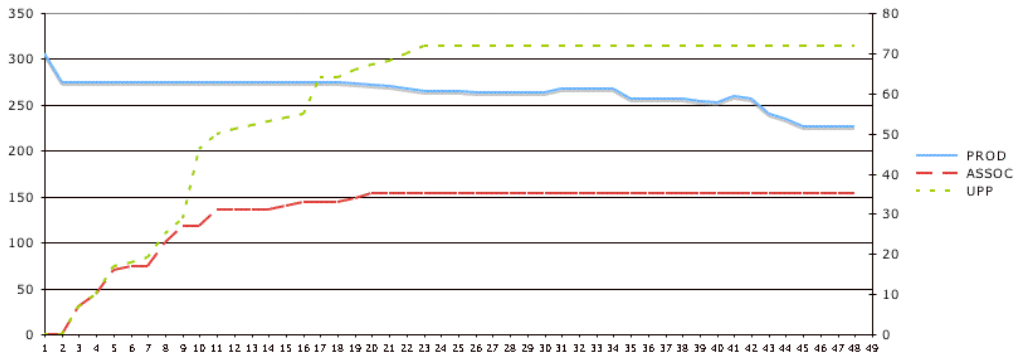


Figure 3.6: The evolution of the ASSOC and UPP ambiguity metrics compared with the evolution of the number of productions (PROD). The x-axis represents the 48 development versions.

Ambiguity metrics

In Figure 3.6 we plot the evolution of two ambiguity metrics and compare them to the number of productions metric (PROD). Although we computed more metrics, we chose to show only the number of associativity attributes (ASSOC) and the number of unique productions in priorities (UPP) because these are the two types of disambiguation information most used during the development.

In the disambiguation phase, 31 development versions were produced (version 32 corresponds to the disambiguated grammar). Yet, by analyzing the chart we can see that the ASSOC and the UPP metrics stabilize after the 23rd version. This is because after this version other kind of disambiguation information was added (reject productions and lookahead restrictions) which are not covered by the chosen metrics.

Also, it is interesting to see that in the 2nd development version no disambiguation information was added but the number of productions drops significantly. This was due to injection removal necessary to prepare for disambiguation.

Between versions 2 and 9, the UPP and ASSOC metrics grow, in most cases, at the same rate. In these steps, the binary expressions were disambiguated using associativity attributes (to remove ambiguity between a binary operator and itself) and priorities (to remove ambiguity between a binary operator and other binary operators). Between versions 9 and 10, a large number of unary expressions were disambiguated, involving priorities (between the unary operator and binary operators) but not associativity attributes (unary operators are not ambiguous with themselves).

From versions 9 to 16 both metrics increase fairly gradually. But in version 17, there is a surge in the number of productions in priorities. This was caused by the simultaneous disambiguation of a group of expressions with somewhat similar syntax (let, def, if, foreach, exits, etc. expressions) which do not have associativity information.

From versions 18 to 24 the number of production in priorities grows simultaneously while the total number of productions decreases. This shows that, once more, disambiguation was only enabled by injection removal or by inlining.

Although not shown in the chart, from the 24th version until the 32th disambiguation was continued by adding lookahead restrictions and reject productions. In this phase lexicals were disambiguated by keyword reservation or by preferring the longer match in lexical recognition. For that reason, the total number of productions remains practically unchanged.

3.2.4 Observations

Some errors were found in the ISO standard, caused by ambiguities between production rules.

Three cases are particular relevant, since changes to the grammar language were made.

Incomplete definition: We noted some omissions in the ISO standard in respect to language keywords. Some of the used language keywords were not specified as reserved, meaning that in our first approach some keywords were incorrectly parsed as identifiers.

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 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 3.2: Grammar metrics for VDM and other grammars. The italicized grammars are in BNF, and their metrics are reproduced from [83]. The remaining grammars are in SDF. Rows have been sorted by Halstead effort (HAL-E), which is reported in thousands.

Syntax Ambiguity: There was an ambiguity between `IsDefinedType-Expression` (which will allow the parser to know if a given variable belongs to a predefined type) and the `Apply` expression (used, for instance, in function application). To solve this ambiguity, a terminal was introduced.

Semantic Ambiguity: There are some rules in which it is not possible to distinguish between `Expression` and `CallStatement` rules. This happen because the syntax of `Apply` and `CallStatement` are almost identical. Looking at the IFAD VDM-SL reference [40] we noticed that in all cases this ambiguity existed in the `CallStatement` was removed leading us to believe that an semantic ambiguity existed.

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 3.2, sorted by the value of the Halstead Effort metric (HAL-E).

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 [83]. 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 grammars 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 allows for 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.

3.2.5 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 deviations from the standard, such as VDM^{++}. We have built an integration test suite by collecting specifications

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 VDM-SL (Peter Gorm Larsen)	380	26%	38%
Exercises formal methods course Univ. do Minho (Tiago Alves)	500	35%	48%
Total	2580	50%	70%

Table 3.3: Integration test suite. The second column gives the number of code lines. The third and fourth columns gives coverage values for the final grammar.

from the internet[1]. A preprocessing step was performed to extract VDM-SL specification code from literate specifications. We manually adapted specifications that did not adhere to the ISO standard.

Table 3.3 lists the suite of integration tests that we obtained in this way. The table also shows the lines of code (excluding blank lines and comments) 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 from the final, refactored grammar.

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 (see conclusion and future work).

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

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

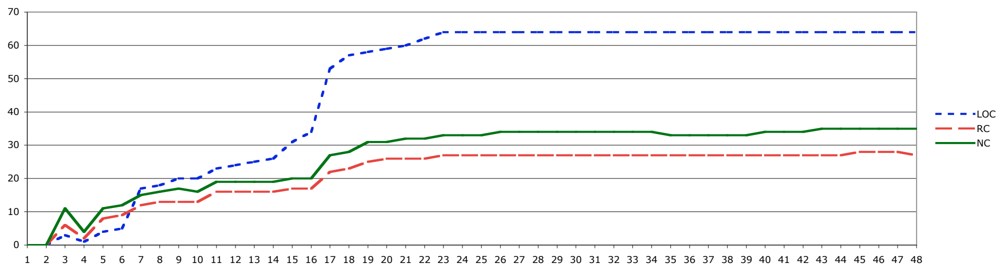


Figure 3.7: The evolution of unit tests during development. The x-axis represents the 48 development versions. The three release versions among these are 1, 32, and 48. The left y-axis corresponds to lines of unit test code. Rule coverage (RC) and non-terminal coverage (NC) are shown as well.

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 3.7. 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, when lexical disambiguation was carried out, unit tests were not added due to limitations of the test utilities.

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

3.3 Summary

In this chapter we document the development of a VDM-SL grammar used as front-end for the VooDooM tool. By choosing the SDF formalism we were able to use a grammar centered approach in which several components are automatically generated, such as the parser, pretty-printer and abstract syntax tree representation. This choice also allowed us to focus on the grammar development and we did so using strong software engineering techniques.

We follow an iterative grammar development approach in which, for every modification made, we apply versioning, metrication (including coverage), and extensive testing using both white box testing and black box testing. We conclude by analysing the evolution of our grammar metrics and comparing these with others in the literature.

Chapter 4

Support for understanding

Program understanding is the process of recovering the knowledge embedded in computer programs. This means abstracting from the source code details which lead to better understanding of what the programmers had in mind when they write a particular piece of code. It can be achieved using ad-hoc techniques, such as manually browsing source code, or by using automatic techniques for program understanding.

In this chapter we are interested in exploring the latter. We focus on automatic techniques source code at different abstraction levels in which, using a similar approach as [46], the information extracted can be represented by finite graphs.

Using graphs offers many advantages. To name a few, graphs can be used to represent different kinds of information, they are easy to transform (eg. using many known algorithms) and they are easy to visualize.

Visualization will be accomplished using Graphviz [35], an open source-tool developed by AT&T which accepts a simple graph description language (DOT) and automatically lays out the output graph in different formats.

Although formal specifications are abstract models which lay much higher than conventional source code (eg. SQL, C, etc.), they can be as large as programs and difficult to understand. We intend to explore program understanding techniques concerning VDM-SL formal specifications. This

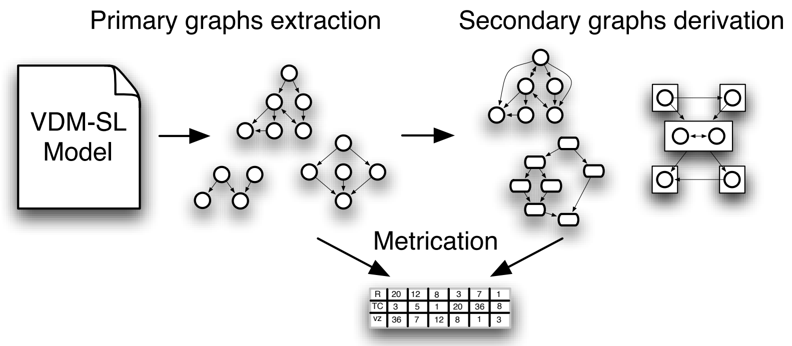


Figure 4.1: VDM-SL understanding process through graph analysis.

will be done in three well defined steps: primary graph extraction, secondary graph derivation, and graph metrication. This three steps are represented in Figure 4.1 and will be explained in each of the following sections.

Section 4.1, will describe how to extract information directly from a VDM-SL specification and how to represent it using graphs.

Firstly, we will introduce a graph library available in the UMinho Haskell Libraries[1]. We will explain how graphs are represented in Haskell and provide a brief overview about the most relevant functions.

Secondly, we will formally define procedure call, type dependency, and procedure-type flow graphs and explain how these graphs can be extracted and challenges involved in the extraction process. Explanations will be enriched with examples and pictures of the resulting graphs.

In Section 4.2 we will introduce graph transformations such as transitive closure, strongly connected components computation and formal concept analysis [103, 102, 101, 37]. These transformations will be applied to the graphs extracted from the VDM-SL specification to obtain different kinds of abstraction views.

Finally, from both the extracted and derived graphs, we will compute structural metrics adding quantification to abstract models.

[1]`http://wiki.di.uminho.pt/wiki/bin/view/PURe/PUReSoftware`

As test case we decided to use the SAFER VDM-SL specification, re-engineered from an existing PVS specification by Sten Agerholm and Peter Gorm Larsen [2, 1]. The SAFER system specifies some important components of a NASA specification of a lightweight propulsive backpack system designed to provide self-rescue capabilities to a NASA space crew-member separated during an extravehicular activity. The SAFER models the thrusters selection based on commands coming from both a hand grip and the existent automatic attitude hold.

For the reader's convenience the SAFER specification, previously published in [2], is in Appendix.

We intend to analyse this specification using our program understanding techniques to get better insight about the system's architecture.

4.1 Primary graphs extraction

4.1.1 Graphs representation

Using the definition found in [15], a directed graph G is as a pair (V, E), where V is a set of vertices, and E is a set of edges between the vertices, $E \subseteq \{(u, v) \mid u, v \in V\}$.

An edge (u, v) captures a relationship between u and v. One says that u is connected to v. Using as example the type dependency graph, the relationship (u, v) means that the globally defined type u is defined using the globally defined type v. For the other dependency graphs, definitions will be provided in the sequel.

In the UMinho Haskell Libraries, graphs are modelled exactly as in their formal definition: a pair of a set of vertices and a relation between the elements of that set.

A relation between types a and b is simply a set of pairs of inhabitants of such types. The corresponding Haskell datatype definitions can be found in Figure 4.2. Note that an invariant should be associated with the `Graph` datatype to record the fact that all elements of the `Gph` component must be

$$
\begin{array}{|l|}
\hline
\quad \textbf{type}\ \mathit{Graph}\ a = (\mathit{Set}\ a, \mathit{Gph}\ a) \\
\quad \textbf{type}\ \mathit{Gph}\ a = \mathit{Rel}\ a\ a \\
\quad \textbf{type}\ \mathit{Rel}\ a\ b = \mathit{Set}\ (a, b) \\
\hline
\end{array}
$$

Figure 4.2: Haskell Graph datatype definition.

contained in the other component.

The simplicity of this datatype is also its main strength. It is very easy to use, specially when collecting information with traversals using strategic term-rewritting, as it will be shown in the sequel.

A comprehensive set of functionalities is already available from the **GraphR** library which makes it the perfect candidate to be used in this work. Figure 4.3 lists the signatures[2] of the functions used in this work.

4.1.2 Procedure call graph extraction

A procedure call graph records the invocation relationship between functions and operations in a formal specification. Recalling the formal definition presented in Section 4.1.1, a procedure call graph is a pair (V, E), in which V is the set of globally defined functions or operations and E is a relation between elements of V. In this context, an edge (p_{caller}, p_{callee}) records the fact that function (or operation) p_{caller} can call function (or operation) p_{callee}.

Computing the procedure call graph is not easy because it must take into account a few details such as higher order functions, locally defined functions and mappings. This is illustrated in the example of Figure 4.4 which shows to subtle details.

In line .2 a lambda function is locally defined using a `let` expression and it is applied to the `m` variable in line .3. Since a lambda function is not a globally defined function, it should not be considered in the procedure call graph.

[2]Some of the datatypes will be defined in the following sections.

```
transClose :: Ord a ⇒ Rel a a → Rel a a
strongComponentGraph :: (Ord a, Ord (Set a)) ⇒ Gph a → Graph (Set a)
lattice :: (Ord g, Ord m) ⇒ Context g m → ConceptLattice g m
printRel :: (Show a, Ord a, Show b, Ord b)
          ⇒ GraphName → Rel a b → String
printRelWith :: (Ord a, Ord b)
                ⇒ (a → NodeName)
                → (Rel a b → a → String)
                → (b → NodeName)
                → (Rel a b → b → String)
                → GraphName
                → Rel a b
                → DotGraph
printConceptLattice :: (Ord g, Ord m)
                       ⇒ (Set g → String)
                       → (Set m → String)
                       → ConceptLattice g m
                       → String
```

Figure 4.3: Overview of the functionality available from the GraphR library.

The second detail has to do with finite mappings (partial functions) passed as arguments. In line .5, we are retrieving the value associated with key `0`. Since a mapping is (mathematically) a partial function, the notation of a map is the same as that of a function. Nevertheless, be it a function or a map, the variable `m` is not a globally defined function and hence should also not be considered in the graph.

The procedure call graph of this code excerpt is simply represented as an edge from `exampleFunction` to `applyFunc`. An example of a procedure call graph will be shown latter.

In the implementation is necessary to consider both locally defined variables and variables passed as function arguments so as not to introduce false

```
$exampleFunction : \mathbb{Z} \xrightarrow{m} (\mathsf{char}^*) \rightarrow \mathsf{char}^*$

$exampleFunction\,(m) \triangleq$
    $\mathsf{let}\ mapIsEmpty = \lambda\, m : \mathbb{Z} \xrightarrow{m} \mathsf{char}^* \cdot \mathsf{card\ dom}\ m = 0\ \mathsf{in}$
    $\mathsf{if}\ mapIsEmpty\,(m)$
    $\mathsf{then}\ applyFunc\,(m)$
    $\mathsf{else}\ m\,(0);$
```

Figure 4.4: VDM function call example with locally defined function.

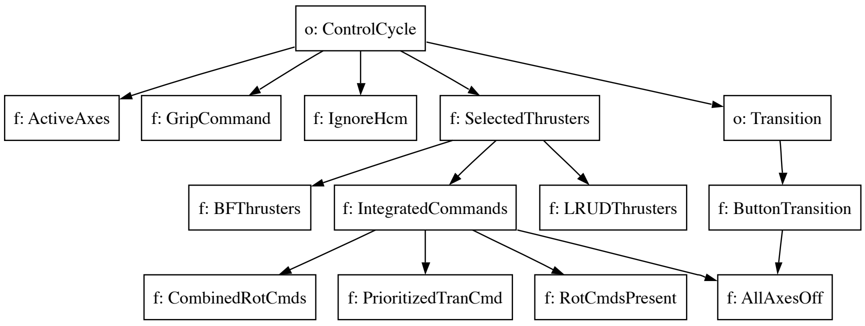


Figure 4.5: Procedure call graph of the SAFER specification.

references into the procedure call graph.

To compute the procedure call graph a strategy was developed to traverse the source code tree and identify function and operation definitions. For each function or operation, a first traversal is performed to collect all parameter names. The variable names will be used as state when collecting all functions or operations called. This is performed by a second traversal function, that updates the variable name list wherever it finds a `let` or a `def` expression, and returns a function or operation application if the function or operation names are not in the list.

The outcome of procedure call analysis of the SAFER specification is given in the graph depicted in Figure 4.5. This and the following graphs has a mark node with tags `t`, `s`, `f`, `o` meaning datatypes, state, functions and operations, respectively.

From this graph we can observe several facts about both structure and system's functionality.

For instance, it reveals small complexity due its resemblance with a tree. (Note that the structure it is not a tree only because both `Integrated-Commands` and `ButtonTranstion` call `AllAxesOff`.)

From the same graph we can also clearly identify the main, or top-level, function of the system, `ControlCycle` and then follow its dependencies. Since only `SelectedThrusters` has dependencies, we can guess that most of the work is done in this function and it dependencies, while `ActiveAxes`, `GripCommand` and `IgnoreHcm` are just auxiliary functions. Moreover, the `Transition` function reveals only two dependencies (one direct and other indirect) which indicates some separation of concerns.

By manually inspecting the specification files we confirm our expectations. Functions `ActiveAxes`, `IgnoreHcm` and `GripCommand` are all auxiliary functions. The first two return state values and the latter is a table lookup function which performs some conversions.

Doing a similar analysis to the `SelectedThrusters` function, we can observe that are also two auxiliary functions `BFThrusters` and `LRUDThrusters`. Inspecting the functions we again confirm our expectations, since both functions perform lookups using "case" statements to select a subset of the available thrusters.

By iteratively following function dependencies we were able to uncover important facts about the system. Not only we were immediately able to identify the main function, but we were also able to discover the functions that perform the most relevant computations.

4.1.3 Type dependency graph extraction

A type dependency graph records the dependency relationship between globally defined datatypes. Recalling the formal definition presented in Section 4.1.1, a type dependency graph is a pair (V, E), in which V is the set

```
2.0  SwitchPositions :: mode : ControlModeSwitch
 .1                     aah : ControlButton;

3.0   ControlModeSwitch = ⟨Rot⟩ | ⟨Tran⟩;

4.0   ControlButton = ⟨Up⟩ | ⟨Down⟩;

5.0  HandGripPosition :: vert : AxisCommand
 .1                      horiz : AxisCommand
 .2                      trans : AxisCommand
 .3                      twist : AxisCommand
```

Figure 4.6: Excerpt of SAFER datatype definitions.

of globally defined datatypes and E is a relation between elements of V. In this context an edge (t_{uses}, t_{isUsed}) is defined if the t_{uses} datatype uses t_{isUsed} datatype in its definition.

In VDM-SL a state definition can also be regarded as a globally defined datatype, it has a name and is defined using other datatypes. Thus, our type dependency graph will also consider a state as an ordinary type.

As example we will consider an excerpt of the SAFER specification presented in Figure 4.6. In this excerpt, we can see two particularities. Firstly there are two unconnected datatypes: `SwitchPositions` and `Hand-GripPosition`; and secondly the `HandGripPosition` datatype references the `AxisCommand` datatype four times. Note that only one edge is created in the graph between `HandGripPosition` and `AxisCommand`.

The result of the type dependency analysis of the SAFER specification is represented in Figure 4.7.

In this figure we can observe that there are three unconnected graphs whose top-level datatype names are `SixDofCommand`, `SwitchPositions`, and `ThrusterSet`.

The graph alone unveils little information about these types. Nevertheless by showing how the datatypes are structured, we can use an approach similar

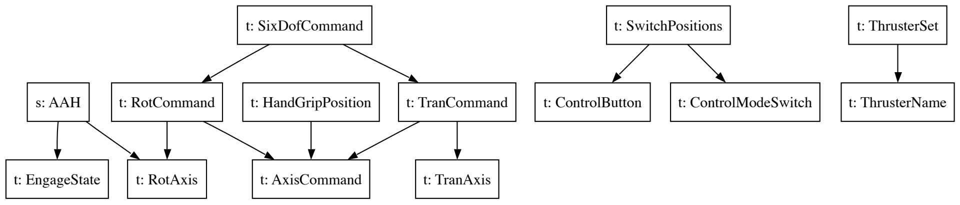


Figure 4.7: Type dependency graph for the SAFER specification.

```
6.0  $GripCommand : HandGripPosition \times ControlModeSwitch \to SixDofCommand$
 .1  $GripCommand$ (mk-$HandGripPosition\,(vert, horiz, trans, twist), mode)$ $\triangleq$
 .2    let ...
 .3        in mk-$SixDofCommand\,(tran, rot)$
```

Figure 4.8: SAFER excerpt function definition.

to that used in the previous section, by starting analysing the top-level datatype and then following all its dependencies.

In contrast with `SixDofCommand` and its dependencies, the `SwitchPositions` and `ThrusterSet` datatypes serve specific details.

The `SwitchPositions` and its dependencies are a dedicated set of datatypes which are defined to model the hand controller module, and the `ThrusterSet` is the result datatype of the main function which contains the set of thrusters that should be activated.

Although the type dependency graph reveals the datatype structure, it is difficult to understand how these datatypes are actually used in the system. For that purpose a more appropriate graph should be produced, such as the procedure-type flow graph.

4.1.4 Procedure-type flow graph extraction

A procedure-type flow graph records the dependency relationship between functions or operations and the globally defined datatypes used as input and output. Recalling the formal definition presented in Section 4.1.1, a procedure-

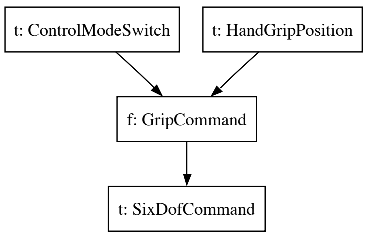


Figure 4.9: SAFER excerpt procedure-type flow graph.

type flow graph will be a pair (V, E), in which V is the set of globally defined functions, operations and datatypes, and E is a relation between them, i.e., an edge (t_{input}, p) is defined wherever the globally defined datatype t_{input} is used as input in the function or operation p, and an edge (p, t_{output}) is defined if the globally defined datatype t_{output} is the result datatype of the function or operation p.

As example we will consider the excerpt from the SAFER specification presented in Figure 4.8. From the function definition we can observe that the `GripCommand` function has as input the `HandGripPosition` and `ControlModeSwitch` datatypes, and has as output the `SixDofCommand` datatype.

The procedure-type flow graph of this code excerpt is represented in Figure 4.9, showing that relationship. Note that, wherever a function uses the same datatype more than once, only one edge will be represented in the graph. Moreover, if the function definition uses `set of ControlModeSwith` or a composite datatype which contains the `ControlModeSwitch` datatype, the result graph will be exactly the same.

Although the example refers to a function and not an operation, it is worth noting that a similar result would be achieved for an operation. However, there is a small detail that must be clarified. Operations in VDM-SL manipulate state, which is explicitly defined in the formal specification. Since all operations manipulate the same state we decided not include it in the procedure-type flow graph. For more about this see conclusions.

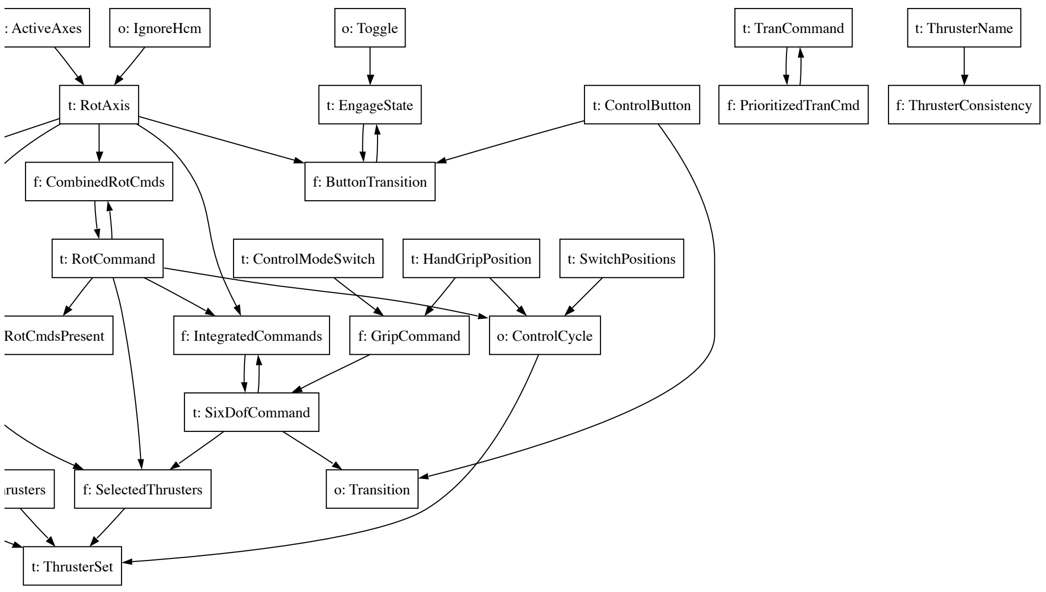


Figure 4.10: Excerpt of the procedure-type flow graph for the SAFER specification.

Procedure-type flow analysis of the whole SAFER specification leads to the graph presented in Figure 4.10.

In the upper right corner of the figure we find two functions that are isolated: `PrioritizedCommand` and `ThrusterConsistency`.

The `ThrusterConsistency` uses the `ThrusterName` datatype that is not used elsewhere. Since we saw, in Section 4.1.3, that `ThrusterName` belongs to the result datatype of the main function, we can assume that this function performs some validation. Analysis of the source code, reveals that this function is the post-condition used in the main function, hence our assumptions are correct.

The `PrioritizedCommand` has both as input and output the `TranCommand` datatype. Additionally, by analysing the procedure call graph, Figure 4.5 of Section 4.1.2, we can see that this function does not depend of any other function. Therefore, we can assume that this function does some sort of transformation. By manually inspecting the function body we find out that it performs corrections by replacing some commands by others with higher priority.

The approach used to analyse the `PrioritizedCommand` function can also be used to analyse `ButtonTransition` and `IntegratedCommands` functions.

Moreover, it is interesting to note that although `ControlCycle` is the main function, there are other functions that have the same output datatype, such as `SelectedThrusters` and `LRUDThrusters`.

The procedure-type flow graph provides a good indication about the abstract flow of the datatypes which can give us hints about what some functions do. Nevertheless to validate those hints we sometimes used information from other graphs and we did manual inspection of the implementation to validate hypotheses.

4.2 Secondary graph derivation

By secondary graph derivation we mean transformation of previously extracted graphs.

Examples of transformations are edge reversal, aggregation of different graphs or applying well known algorithms such as transitive closure, strongly connected components and formal concept analysis.

Each of these transformations will be explained in the following sections.

4.2.1 Procedure-type dependency derivation

A procedure-type dependency graph is an abstraction of the procedure-type flow graph presented in Section 4.1.4 in which it is irrelevant whether if the datatype is used as input or output.

Recalling the formal definition presented in Section 4.1.1, a procedure-type dependency graph is a pair (V, E), in which V is the set of globally defined functions, operations and datatypes, and E is a relation between functions or operations, and datatypes, i.e., and edge (p_{uses}, t_{isUsed}) is defined

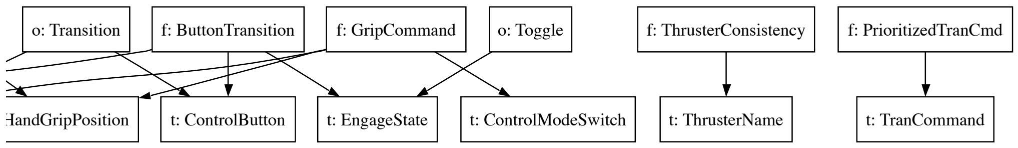


Figure 4.11: Excerpt of the procedure-type dependency graph for the SAFER specification.

if the procedure (function or operation) p_{uses} uses the t_{isUsed} datatype in the procedure definition.

This graph is derived from the procedure-type flow graph by reversing all edges (t, p), in which t is a globally defined datatype and p is a globally defined function or operation.

By applying this analysis to the SAFER specification one obtains a huge graph, of which only the right part is represented in Figure 4.11.

Although this analysis produces more abstract information than the procedure-type flow graph, introduced in Section 4.1.4, we are not able to extract more knowledge than using that graph. We are able to visualize the procedures that are unconnected, namely `ThrusterConsistency` and `PrioritizedTranCmd`, yet for the rest of the graph interpretation is burdensome.

This outcome of the analysis does not mean that it is inadequate for understanding. What it means ist that for this particular specification does not add much value. With larger specifications the scenario can be different. For more of this see conclusions.

Additionally, it is possible to feed this graph to formal concept analysis to try and find groups of functions that use the same types, thus providing us, for instance, with some modularization hints.

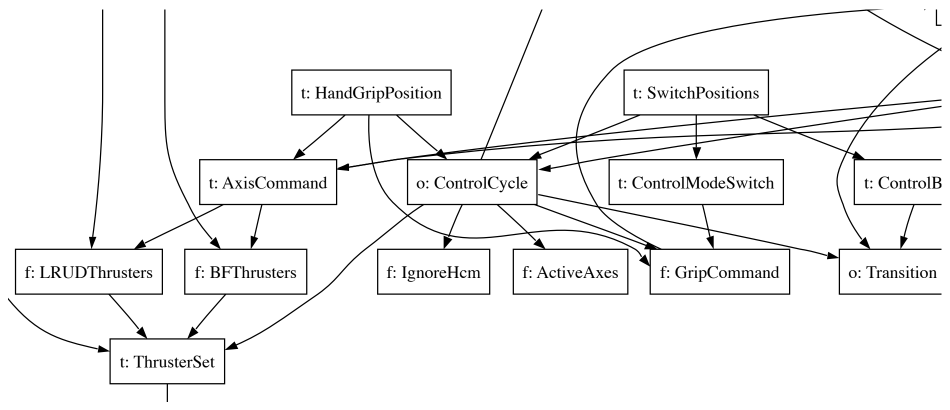


Figure 4.12: Excerpt of the full dependency graph for the SAFER specification.

4.2.2 Full dependency graph derivation

The full dependency graph is a union of the procedure call, type dependency, and procedure-type flow graphs defined, respectively, in Sections 4.1.2, 4.1.3, and 4.1.4.

Recalling the formal definition presented in Section 4.1.1, a full dependency graph is a pair (V, E), in which V is the set of globally defined functions, operations and datatypes, and E is a relation between elements of V, such as:

- (p_{caller}, p_{callee}) is defined if the function or operation p_{caller} can call p_{callee},
- (t_{uses}, t_{isUsed}) is defined if the t_{uses} datatype uses t_{isUsed} datatype in its definition,
- (t_{input}, p) is defined if the t_{input} datatype is used as input in the function or operation p, and
- (p, t_{output}) is defined if the t_{output} datatype is the result datatype of the function or operation p.

The result of the full dependency analysis of the SAFER specification is represented in Figure 4.12. It can be observed that, even for a small

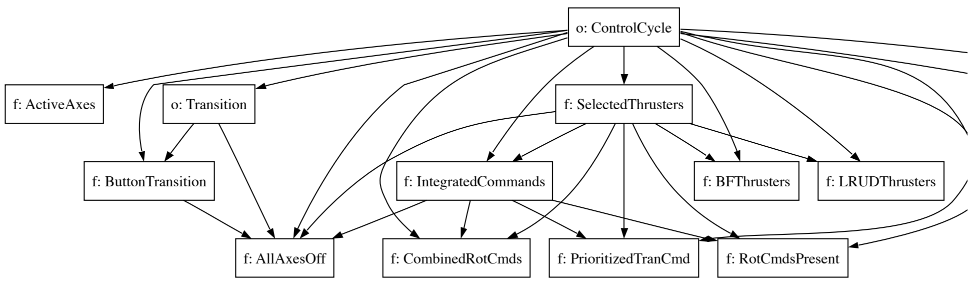


Figure 4.13: Excerpt of the Function and Operation transitive closure call graph for the SAFER Specification.

specification like the one we chose, it is not very easy to extract knowledge from it.

Still, this kind of graph is a perfect candidate for applying the strongly connected components analysis and discover groups with related functionalities, as we will see in further sections.

4.2.3 Transitive closure derivation

The transitive closure (TC), as defined in [103], of a binary relation R on a set X is the minimal transitive relation R^+ on X that contains R and it can be defined as $R^+ = R \cup R.R^+$. Thus $uR^+v \equiv uRv \vee \exists\, w \cdot, uRw \wedge wR^+v$, meaning that the transitive closure of graph, is a graph which contains an edge (u, v) whenever there is a direct or indirect path from u to v.

Figure 4.13 represents the transitive closure graph of the procedure call graph of Figure 4.5.

The transitive closure can be used for reachability analysis, i.e., it allows for querying, in a efficient way, if any two nodes on a graph are connected. This can be used to detect dependencies that are not immediately perceived and/or to detect cycles.

In case of the transitive closure of the procedure call graph, it can also be used to guide testing, in the sense that nodes that are dependent of many nodes should be paid particular attention. In the graph of Figure 4.13 we can observe that the `AllAxesOff`, `CombinedRotCmds`, `PrioritizedTranCmd` and `RotCmdsPresent` have the highest rate of dependability.

Although not shown, we also computed the transitive closure for type dependency graph, procedure-type flow graph and full dependency graph. In case of the latter, the graph presents so many edges that is impossible to understand anything at all.

In general, as understanding technique, the transitive closure may present two problems. Firstly, to many dependencies hinder analysis. Secondly, for large specifications the analysis does not scale well. Nevertheless, the transitive closure has practical use in this work, in which the type dependency transitive closure is used during re-engineering of a formal specification to an SQL scheme to discover mutually recursive definitions. For more details will be provided in Section 5.2.1.

4.2.4 Strongly connected components derivation

A strongly connected component (SCC), as defined in [102], is a maximal subgraph of a directed graph such that for every pair of vertices u, v in the subgraph, there is a directed path from u to v and a directed path from v to u.

Informally, the SCC is a graph in which nodes are formed by sets of the directed graph nodes (components) and edges represents the relations between these components. Real-world applications of SCC are partitioning a problem into smaller problems and hence it is used in many fields such as biology or computer science.

We shall be interested in using strongly connected components analysis to reveal the architecture or structure of the analysed data, in which as result we will get several nodes aggregated in many components. Components can be regarded as programming language modules which can provide us hints about possible ways of modularizing, for instance, functions.

However, SCC analysis can lead to two extreme situations: either the graph results in a single component (meaning that all elements are connected

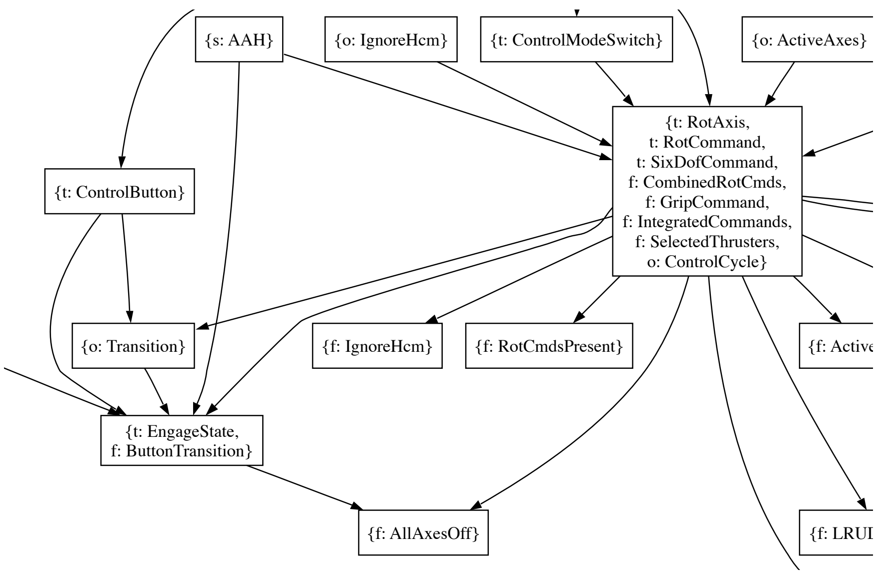

{s: AAH}
{o: IgnoreHcm}
{t: ControlModeSwitch}
{o: ActiveAxes}
{t: RotAxis, t: RotCommand, t: SixDofCommand, f: CombinedRotCmds, f: GripCommand, f: IntegratedCommands, f: SelectedThrusters, o: ControlCycle}
{t: ControlButton}
{o: Transition}
{f: IgnoreHcm}
{f: RotCmdsPresent}
{f: Active
{t: EngageState, f: ButtonTransition}
{f: AllAxesOff}
{f: LRUI

Figure 4.14: Excerpt of the Strongly Connected Components for the full dependency graph.

to each other) or each element ends up forming its own component, indicating that no grouping took place. The latter case could be observed in both the procedure call and type dependency graphs.

Applying the strongly connected components analysis to the full dependency graph results in the graph partially presented in Figure 4.14.

Only the full dependency and procedure-type flow graphs present cycles, being the only graphs in which the strongly connected components analysis would do some grouping. We chose the full dependency graph since it is the one containing more information.

By analyzing this graph, we can identify three non-singleton components (components which have more than one element grouped). Two of these components are small, containing only two elements. The other is thus a quite large component.

Starting from the right hand side of the graph, we can see the component formed by the `ButtonTransition` function and the `EngageState` datatype. This indicates that both should belong to the same module in the specification.

Moreover, all around that component there exist singleton components, such as the `Toggle` datatype and `Transition` and `AllAxesOff` functions, which are related to this component and could also be included in the same module.

The larger component is much more interesting to analyse since it groups togehter the most important functions and datatypes of the system. We can start by observing that the `ControlCycle` main function is present in this component. Additionally, functions previously identified as important, such as `SelectedThrusters` and `IntegratedCommands`, are also in the same component, as well as the involved datatypes.

It is interesting to note that SCC analysis identifies the same set of important functions as with other techniques, namely the procedure call and the procedure-type flow graphs.

Even so our testcase is small, these techniques reveal to be of interest. We guess that with larger specifications we would first apply the SCC analysis first and then do a more fine grain analysis to each component using the other techniques.

4.2.5 Formal Concept Analysis derivation

Formal Concept Analysis (FCA) is a mathematical theory which detects the presence of groups of elements which instantiate a common, repeated pattern [19]. FCA was firstly introduced by Wille [105, 37] based on Lattice Theory, the foundations of which were laid in 1967 by Birkhoff [13]. For a more extensive background about FCA, we refer the reader to [105, 37, 19].

Before studying the applicability of FCA to our graphs (eg. procedure call graph) we will firstly introduce the basic concepts of theory.

The first step in FCA is to define a *context*. A *context* $\mathcal{C}$ is formally defined as a triple $\mathcal{C} = (\mathcal{E}, \mathcal{P}, \mathcal{I})$, in which $\mathcal{E}$ is a finite set of *elements*, $\mathcal{P}$ is a finite set of *properties* and $\mathcal{I}$ is a binary relation between $\mathcal{E}$ and $\mathcal{P} : \mathcal{I} \subseteq \mathcal{E} \times \mathcal{I}$.

Based on the formal definition of *context*, two mapping relations can be defined: *common properties of* X and *common elements of* Y, respectively represented as $\sigma(X)$ and $\tau(Y)$, where $X \subseteq \mathcal{E}$ and $Y \subseteq \mathcal{P}$. Then:

```
type Context g m = Rel g m
type Concept g m = (Set g, Set m)
type ConceptLattice g m = Rel (Concept g m) (Concept g m)
```

Figure 4.15: Haskell datatype definitions to support FCA.

$$\sigma(X) = \{p \in \mathcal{P} \mid \forall e \in X : (e, p) \in \mathcal{I}\}$$

$$\tau(Y) = \{e \in \mathcal{E} \mid \forall p \in Y : (e, p) \in \mathcal{I}\}$$

Informally, $\sigma(X)$ defines the set of common properties of the elements of X, and $\tau(Y)$ defines the set of elements that share the properties in Y.

Based on the formal definitions of $\sigma(X)$ and $\tau(Y)$, a *concept* is a maximal collection of elements sharing common properties and can be formally defined as a pair of sets (X, Y): a set of elements (the *extent*) and a set of properties (the *intend*), such that:

$$Y = \sigma(X) \text{ and } X = \tau(Y)$$

Finally, we can define a *concept lattice* as the ordered set of all *formal concepts*. Informally, this can be defined as a graph which represents the relation between all *concepts*.

The Haskell datatypes definition for *context*, *concept* and *concept lattice* are represented in Figure 4.15.

FCA, as contrast with other program understanding techniques, does not summarize information. It has all the details of the data represented by the formal context. As stated in [36], the typical task of a concept lattice is to unfold some given data, making the conceptual structure visible and accessible in order to find patterns, regularities, exceptions, etc.

As test case we will analyze the results of applying formal concept analysis to the procedure call graph presented in Section 4.1.2.

	ActiveAxes	AllAxesOff	BFThrusters	ButtonTransition	CombinedRotCmds	ControlCycle	GripCommand	IgnoreHcm	IntegratedCommands	LRUDThrusters	PrioritizedTranCmd	RotCmdsPresent	SelectedThrusters	Transition
ActiveAxes														
AllAxesOff														
BFThrusters														
ButtonTransition		x												
CombinedRotCmds														
ControlCycle	x						x	x					x	x
GripCommand														
IgnoreHcm														
IntegratedCommands		x			x						x	x		
LRUDThrusters														
PrioritizedTranCmd														
RotCmdsPresent														
SelectedThrusters			x						x	x				
Transition				x										

Table 4.1: Formal context of the procedure call relationship of the SAFER model identifying two concepts marked in gray.

The first step is the definition of the formal context of the procedure call relationship which is obtain from the procedure call graph, represented in Figure 4.1.

Lines and columns represent, respectively, p_{caller} and p_{callee}, i.e., there is an x in line p_{caller} and column p_{callee} if the function or operation p_{caller} calls the function or operation p_{callee}.

From this formal context, the computed concept lattice graph is presented in Figure 4.16.

We can observe that the concept lattice has a total of five concepts. Two of them are related and the other three are completely independent of each other, and of the first two.

The two concepts that are related were previously marked in gray in the formal context table of Figure 4.1.

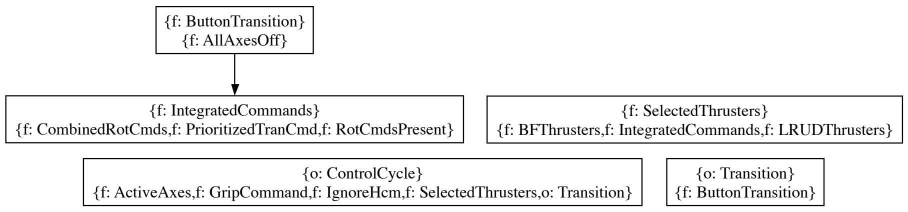

{f: ButtonTransition}
{f: AllAxesOff}
{f: IntegratedCommands}
{f: CombinedRotCmds,f: PrioritizedTranCmd,f: RotCmdsPresent}
{f: SelectedThrusters}
{f: BFThrusters,f: IntegratedCommands,f: LRUDThrusters}
{o: ControlCycle}
{f: ActiveAxes,f: GripCommand,f: IgnoreHcm,f: SelectedThrusters,o: Transition}
{o: Transition}
{f: ButtonTransition}

Figure 4.16: Excerpt of the concept lattice for the procedure call relationship of the SAFER specification.

In the fourth line of the table, marked with lighter gray, the concept is formed by the `ButtonTransition` element and the `AllAxesOff` attribute. In the ninth line, marked in darker gray, the concept is formed by the `IntegratedCommands` element and four attributes: `AllAxesOff`, `Combined-RotCommands`, `PrioritizedTranCmd`, and `RotCmdsPresent`. Note that, because the two elements share the same `AllAxesOff` attribute, the two concepts are related. However, the former concept is considered a *parent-concept* of the latter *child-concept*, since the latter has attributes that do not exist in the former.

We must shed light on an important detail about the concept lattice graph presented in Figure 4.16. As the `AllAxesOff` attribute is shared by two concepts, this attribute only appears in the *parent-concept* and not in the *child-concept*, since all *child-concepts* inherit all the *parent-concept*'s attributes.

By analysing the concept lattice it is interesting to observe that there are unconnected concepts as previously mentioned. By individually analysing the concepts, we observe that the most important set of functionalities is represented in three concepts.

The `ControlCycle` function, which here represents a concept, was identified in Section 4.1.2 as the main function. In the same section, we have also previously identified that the overall functionality of the system was specified in two other functions, `SelectedThrusters` and `IntegratedCommands`, which also are represented in their own concepts.

The relationship between the *parent* and *child* concepts was already explained before. The concept lattice indicates us that the `ButtonTranstion` concept is a more abstract concept than the `IntegratedCommands` concept. However, by manually inspecting the body of the functions we realize that the `ButtonTranstion` operation is responsible for activating or deactivating the automatic attitude hold using a specific protocol, and the `Integrated-Commands` function is responsible for aggregating commands information, meaning that the two functions are not semantically related.

We can hypothesize two alternative explanations for this. Either the formal concept analysis can sometimes be misleading, or the SAFER test case is too small for a more accurate interpretation.

4.3 Metrics

Software metrics are an important tool for both controlling and validating software development. With respect to their computation, metrics can be classified as primitive or computed. A primitive metric is calculated directly from the source code while the second is computed using other metrics. As discussed in [72], computed metrics can provide a more general overview of the aspects that are measured, since they aggregate information from other metrics.

Many different metrics exist with different purposes. For instance, the LOC metric (source lines of code) measures size, the McCabe metric measures complexity, Function Point metric measures productivity, and so on.

In this work we are particular interested in structural metrics that are defined over graph representations and we intend to use them as yet another technique for program understanding. The metrics were adapted from grammar engineering metrics previously introduced in [83, 97, 7].

This section is divided in two parts. Section 4.3.1 provides the definition of several structure metrics, introduced in [97, 8], and adapts them to formal specifications whenever necessary. Section 4.3.2 we will apply those metrics

Structure metrics	
Tree impurity after trans. closure	(%)
Tree impurity of immediate successors	(%)
Fan-in	(absolute)
Fan-out	(absolute)
Instability	(%)
Efferent coupling	(absolute)
Afferent coupling	(absolute)
Instability	(average %)
Coherence	(average %)
Normalized count of modules	(%)

Figure 4.17: Structure metrics for VDM-SL formal specifications.

and provide some observations about the results obtained.

As in [83, 97, 7, 8] a rigorous empirical validation of these metrics falls outside the scope of this work.

4.3.1 Metrics Definition

In this section we will introduce a set of metrics and present their definition. A summary indicating whether the metric uses absolute values or percentages, can be found in Table 4.17.

Tree Impurity

The tree impurity metric (TIMP) indicates to what extent a given graph deviates from a tree structure with the same number of nodes. Fenton *et al.* [30] define tree impurity for connected undirected graphs without self-edges as $\frac{2(e\text{-}n+1)}{(n\text{-}1)(n\text{-}2)}.100\%$, where n is the number of nodes, and e is the number of edges. A tree impurity of 0% means that the graph is a tree and a tree impurity of 100% means that it is a fully connected graph.

Power *et al.* [83] apply the tree impurity metric to the *transitive closure* of the immediate successor graph, i.e. to the (non-immediate, transitive) successor graph. Since tree impurity is defined for undirected graphs without self-edges, this requires that self-edges and multiple edges between non-terminals should be removed from the edge count before applying the formula.

In several situations, where we ignore edge direction, a path can exist between any two edges, leading to exaggerated values of the tree impurity metric. As seen in [8], even purely tree-shaped graphs will drastically increase in tree impurity when we take their transitive closure.

Hence, we propose to apply the tree impurity metric also to the *immediate* successor graph, and we will call this metric TIMPi.

Fan-in, Fan-out, and Instability

A pair of classic metrics are *fan-in* and *fan-out*. The fan-in of a node in a directed graph is the number of its incoming edges. Conversely, the fan-out is the number of outgoing edges of the node. Both metrics are directly applicable to the nodes of each type of dependency graph. For the graphs as a whole, the average and maxima of these metrics can be relevant.

The maximum fan-in and fan-out, in particular, can be useful to spot unusual nodes, which subsequently may be inspected to identify the cause of the abnormality, and a possible action to correct the situation.

Based on fan-in and fan-out, a measure called *instability* can be defined as the fan-out fraction of total fan-out, i.e. as: $\frac{\textit{fan-out}}{\textit{fan-in}+\textit{fan-out}}.100\%$.

The instability metric ranges between 0% (no outgoing edges) and 100% (only outgoing edges). Low instability of a node indicates that it is dependent on few other nodes, while many nodes are dependent on it. Thus, low instability corresponds to a situation where changes to the node will affect relatively many other nodes, and would hence be costly or difficult. In other words, instability may be interpreted as resistance to change.

Afferent and efferent coupling, and again instability

Coupling is a notion similar to fan, but taking modules into account, where any group of nodes may be viewed as a module. The number of edges from nodes outside the module to nodes inside the module is called *afferent coupling* (C_a). Conversely, the number of edges from nodes inside the module to nodes outside is called *efferent coupling* (C_e). Their sum is simply *coupling.* As in the case of fan-in and fan-out, an instability metric can be defined based on afferent and efferent coupling, as: $\frac{C_e}{C_e+C_a}.100\%$.

Since we do not support the IFAD module extension [40], the coupling metrics will be calculated using the derived strongly connected components graphs.

Coherence

Whereas coupling assesses the connections of a module with external nodes, the *coherence* of a module concerns the degree to which its internal nodes are connected with each other. As a general coherence metric we will use the ratio of internal edges of a module versus all edges that start and/or end in a node inside the module. Thus, $C_h = \frac{C_i}{C_i+C_a+C_e}.100\%$, where C_i is the number of edges between nodes inside the module, i.e. the internal edge count. In the limit case of singleton modules, we set $C_h = 100\%$, rather than 0%, expressing that singletons are fully coherent.

Note that this measure of coherence takes external edges into account, not only internal edges. Assuming a stable node count for a module, its coherence increases both with the addition of internal edges and with the removal of external edges. In other words, coherence is compromised both by lack of connections between internal nodes, and by too many connections to the outside, i.e. by breaches of encapsulation.

As an alternative measure of coherence, one may use tree impurity, as defined before, at the module level.

Normalized count of modules

For each notion of module, we can define a normalized count of modules by expressing the module count as a ratio of *potential* module count, which is the number of nodes in the underlying dependency graph. Thus, $NCM = \frac{\#M}{\#N}.100\%$. Thus, the more nodes get grouped into modules, the lower the normalized count of modules. A value of 100% indicates that no grouping has occurred, i.e. each node sits in a separate module (full fragmentation). A value approaching 0% indicates that nodes are grouped together in a very small number of modules (monoliths).

The interpretation of normalized count of modules depends on the underlying notion of module. In case of the notion of modules as strongly connected components, (mutual) recursion gives rise to non-singleton modules. Correspondingly, NCM is a measure of *recursiveness*, where low NCM indicates a high degree of mutual recursiveness. In the case of the notion of modules as global definitions/declarations, local elements give rise to non-singleton modules. Correspondingly, NCM is a measure of *encapsulation*, where low NCM indicates a high degree of encapsulation.

Since we do not support the IFAD module extension, we will only use NCM as a measure of *recursiveness*.

4.3.2 Data Collection

We decided to apply the metrics to the full dependency graph presented in Section 4.2.2. This graph was chosen since it is among the graphs on which we could take advantage of the strongly connected analysis, and hence observe all defined metrics.

The outcome of metrics calculation of the SAFER model is presented in Table 4.2. For metrics which calculate absolute values, the minimum, average and maximum values are presented.

The values present in the table lead to the following observations.

The tree impurity metric of the immediate successors presents a reasonable

Structure metrics			
Tree impurity after trans. closure	72.91%		
Tree impurity of immediate successors	7.14%		
Fan-in	0	2.07	4
Fan-out	0	2.07	7
Instability based on fan	46.57%		
Afferent coupling	0	1.89	5
Efferent coupling	0	1.89	14
Instability based on coupling	45.19%		
Coherence	90.10%		
Normalized count of modules	63.33%		

Table 4.2: Metrics data collection for the full dependency graph of the SAFER specification.

value of 7.14%. This suggests that trying to follow dependencies when doing manual inspection can be feasible. However, the tree impurity metric after transitive closure, as contrast, presents a very high value of 72.91% indicating that most of the nodes are connected.

The fan-in and fan-out metrics reveal that, on average, each node has both 2 incoming and outgoing edges. Since this graph contemplates procedures and datatypes this value suggests that the dependencies are quite small. Moreover, the maximum value for the fan-out metric is only 7, which corresponds to the `RotCommand` datatype. In Sections 4.1.3 and 4.1.4, we identified a set of auxiliary datatypes which were used in the functions and operations that do most of the computations, and the `RotCommand` was one of those datatypes. Since this is the datatype that has more dependencies, we can conclude that this datatype has a great impact in the system.

An instability with a value of 46.57% indicates that nodes are well balanced by having the same number of both incoming and outgoing arrows, as we noted before.

Since the efferent and afferent coupling metrics are analogous to fan-in and fan-out but for modules, and the values obtained are similar, then observations are alike. Hence, by analysing the afferent and efferent coupling we can observe that in average each module has both 2 incoming and outgoing edges.

However, we should note that the maximum efferent coupling is 14, meaning that there is a module with a large number of outgoing arrows. Additionally, we can observe that both afferent coupling and fan-in present higher values than both efferent coupling and fan-out.

The instability based on coupling, like the instability based on fan, has a value of near 50%, meaning that modules are well balanced.

The normalized count of modules indicates a recursiveness of 63.33%. We observed in Section 4.2.4 that the overall system's functionality belong to the largest component found. Moreover, SCC analysis reveal that, except for two components formed with two elements, all others are singletons, i.e., most of the components are formed by a single node. Therefore, we can conclude that the core of the system represents about 63.33% of the overall.

4.4 Summary

In this chapter we applied automatic techniques for program understanding to formal models written in VDM-SL notation. To create abstract representations from VDM-SL specifications we started by extracting procedure call, type dependency, and procedure-type flow graphs. We have shown how inspection of these graphs answers specific questions about the system's details.

Other kinds of graph were derived by aggregating information from several extracted graphs and through transformation such as transitive closure, strongly connected components and formal concept analysis. We analysed those graphs and commented on their applicability for our test case.

Finally, we introduce structure metrics as yet another way of model understanding and we applied them to the full dependency graph of our test case.

Chapter 5

Support for re-engineering

Runnable formal specifications, like the ones written in VDM-SL, cannot be regarded as final software products because they are not efficient enough (they lack in efficience what they gain in abstraction). Therefore, there is a need to lower the level of abstraction while preserving the original model semantics. This kind of re-engineering, wich is known as data-refinement [52], can be carried out by transformation of both data models and the associated functionality.

A particular relevant target is the relational model defined by Codd [24]. Since his pioneering work, relational database theory has been thoroughly studied [68, 91, 38]. At the heart of this we find *normalization*, a theory whereby efficient collections of (relational) files are derived from the original design, which can be encoded in a data-processing language such as SQL [34].

Functional dependency theory and normalization deviate from standard model-oriented formal specification and reification techniques [52, 32]. In the latter, designs start from abstract models which are abstract enough to dispense with normalization. Does one arrive at similar database designs by using data reification techniques?

References [78, 79, 80, 81] address a formal calculus which has been put forward as an alternative to standard normalization theory, by framing database design into the wider area of data refinement [52]. Data models, such as described by E-R diagrams, for instance, are turned into systems of equations involving set-theoretic notions such as finite mappings, sets, and sequences. Integrity constraints and business rules are identified with abstraction invariants [73] and datatype invariants [52], respectively, whose structural synthesis (analysis) by calculation is at the core of the calculus.

In this chapter, we will describe a database schema calculator which, inspired by [81], infers SQL relational meta-data from abstract data models specified in the ISO standard VDM-SL formal modelling notation [32]. In Section 5.1 the theory in which the re-engineering process is based on is introduced, and in Section 5.2, the link from theory to practice will be made explicit by explaining both design and implementation.

5.1 Database design by calculation

The calculation method which underlies our VDM-SL to SQL conversion finds its roots in a "data refinement by calculation" strategy which originated in [78, 79] and has been focussed on relational database design more recently [80, 81]. Reference [76] describes its application to reverse engineering legacy databases.

5.1.1 Abstraction and representation

The calculus consists of inequations of the form $A \leqslant B$ (read: *"datatype B implements, or refines datatype A"*) which abbreviates the fact that there is a surjective, possibly partial function $A \xleftarrow{F} B$ (the *abstraction relation*) and an injective, total relation $A \xrightarrow{R} B$ (the *representation relation*) such that

$$F \cdot R = id_A \tag{5.1}$$

holds, where id_A is the identity function on datatype A. (F is traditionally referred to as a *retrieve* function [52].) Since the equality $R = S$ of two relations R and S is bi-inclusion $R \subseteq S \wedge S \subseteq R$, we have two readings of equation (5.1): $id_A \subseteq F \cdot R$, which ensures that every inhabitant of the abstract datatype A gets represented at B-level; and $F \cdot R \subseteq id_A$, which prevents "confusion" in the representation process:

$$\langle \forall\ b \in B, a \in A\ :\ b\ R\ a :\ \langle \forall\ a' \in A\ :\ a'\ F\ b :\ a' = a \rangle\rangle$$

("Never forget whom you are representing".)

Below we will present a series of particular $\leqslant$-equations which together specify a data model refinement calculus. The types of the refinement relations will be mapped onto rewrite rules in the implementation.

5.1.2 Preorder

It can be shown that $\leqslant$ is a preorder, reflexivity meaning that any datatype represents itself ($R = F = id$) and transitivity meaning that $\leqslant$-steps can be chained by sequentially composing abstractions and representations:

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "F"] \end{tikzcd} \wedge \begin{tikzcd} B \arrow[r, bend left, "S"] \arrow[r, phantom, "\leqslant"] & C \arrow[l, bend left, "G"] \end{tikzcd} \Rightarrow \begin{tikzcd} A \arrow[r, bend left, "S \cdot R"] \arrow[r, phantom, "\leqslant"] & C \arrow[l, bend left, "F \cdot G"] \end{tikzcd}$$

This suggests that one may *calculate* implementations from specifications

$$Spec = X \leqslant X' \leqslant X'' \leqslant \cdots \leqslant Imp$$

by adding implementation *details* in a controlled manner. This also makes sense wherever the representation of a parameter of a datatype needs to be promoted to the overall parametric datatype by *structural data refinement*:

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "F"] \end{tikzcd} \Rightarrow \begin{tikzcd} \mathsf{F}\,A \arrow[r, bend left, "\mathsf{F}\,R"] \arrow[r, phantom, "\leqslant"] & \mathsf{F}\,B \arrow[l, bend left, "\mathsf{F}\,F"] \end{tikzcd} \qquad (5.2)$$

where F is such a parametric type, e.g. `set of` A in VDM-SL notation. (Technically, F is named a *relator* [10].) This is valid also for parametric types of higher arity, such as those of standard VDM-SL:

- binary product types $A \times B$ and n-ary ones $\prod_{i=0}^{n} A_i$, which can be specified in VDM-SL as (nested) tuples or via record types, (semantically equivalent modulo selectors). E.g. `A*B` or `compose AB of a: A b: B end`, respectively.

- sum types $A + B$, which in VDM-SL are specified by writing `A | B` for suitably specified (disjoint) A and B, extensible to finitary sums $\sum_{i=0}^{n} A_i$.
- finite mappings $A \rightharpoonup B$, written `map` A `to` B in VDM-SL, in which case the abstraction of the domain datatype is required to be injective (otherwise the outcome may not be a mapping).

5.1.3 Conversion laws

It is often the case that the abstraction (resp. representation) relation is a (total) function, in which case it is an *injection* (resp. *surjection*). As an example of this we present law

$$\begin{tikzcd} A^\star \arrow[r, bend left, "seq2index"] \arrow[r, phantom, "\leqslant"] & I\!N \rightharpoonup A \arrow[l, bend left, "list"] \end{tikzcd} \tag{5.3}$$

which indexes a finite sequence, for instance,

$$\begin{aligned} seq2index([a, b, a]) &= \{1 \mapsto a, 2 \mapsto b, 3 \mapsto a\} \\ list(\{11 \mapsto a, 12 \mapsto b, 33 \mapsto a\}) &= [a, b, a] \end{aligned}$$

A more structural law is

$$\begin{tikzcd} A \rightharpoonup (B + C) \arrow[r, bend left, "uncojoin"] \arrow[r, phantom, "\leqslant"] & (A \rightharpoonup B) \times (A \rightharpoonup C) \arrow[l, bend left, "cojoin"] \end{tikzcd} \tag{5.4}$$

whereby *mappings of sums* are represented as *products of mappings*. (Definitions for *cojoin* and *uncojoin* are easy to guess.) In a situation where the abstraction is also a representation and vice-versa we have an isomorphism $A \cong B$, a special case of the $\leqslant$-law which works in both directions. For example, the abstraction/representation pair of the following isomorphism

$$\begin{tikzcd} A \times (B + C) \arrow[r, bend left, "distr"] \arrow[r, phantom, "\cong"] & (A \times B) + (A \times C) \arrow[l, bend left, "undistr"] \end{tikzcd} \tag{5.5}$$

(product distributes through sum) is well-known from set-theory.

The VDM-SL finite mapping *dom* function witnesses a very useful isomorphism between finite sets and partial finite mappings,

$$\begin{tikzcd} 2^A \arrow[r, bend left, "set2fm"] \arrow[r, phantom, "\cong"] & A \rightharpoonup 1 \arrow[l, bend left, "dom"] \end{tikzcd} \tag{5.6}$$

which expresses the equivalence between VDM-SL data models `set of A` and `map A to nil`. (The inhabitants of $A \rightharpoonup 1$, often called *right-conditions* [47], obey a number of interesting properties.) Another basic isomorphism tells us how “singleton” finite mappings disguise “pointers” (*opt-intro* and *opt-elim* are easy to guess):

$$\begin{tikzcd} A+1 \arrow[r, bend left, "opt\text{-}intro"] \arrow[r, phantom, "\cong"] & 1 \rightharpoonup A \arrow[l, bend left, "opt\text{-}elim"] \end{tikzcd} \tag{5.7}$$

The following isomorphism law

$$\begin{tikzcd} (B+C) \rightharpoonup A \arrow[r, bend left, "unpeither"] \arrow[r, phantom, "\cong"] & (B \rightharpoonup A) \times (C \rightharpoonup A) \arrow[l, bend left, "peither"] \end{tikzcd} \tag{5.8}$$

is a companion of (5.4).

Two important $\leqslant$-rules from [81] are still missing from our catalog: *unnjoin*, the representation function of one of these,

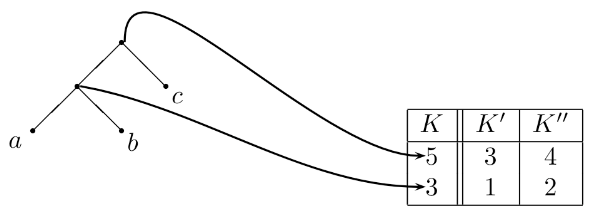
$$\begin{tikzcd} A \rightharpoonup (B \times (C \rightharpoonup D)) \arrow[r, bend left, "unnjoin"] \arrow[r, phantom, "\leqslant"] & (A \rightharpoonup B) \times (A \times C \rightharpoonup D) \arrow[l, bend left, "njoin"] \end{tikzcd} \tag{5.9}$$

enables us to infer *composite keys* out of nested finite mappings. (See [79, 80] concerning abstraction *njoin* and representation *unnjoin*.) In the abstraction direction (from right to left) it merges two tables which share a common (sub)key.

The other rule still missing has to do with datatype "derecursivation". Suppose we are given a recursive datatype definition $\mu\mathsf{F} \cong \mathsf{F}\,\mu\mathsf{F}$ where F is polynomial [10, 79]. Then any "tree" in $\mu\mathsf{F}$ can be represented by a "heap" and a "pointer" to it,

$$\begin{tikzcd} \mu\mathsf{F} \arrow[r, bend left, "\mathit{rec\text{-}elim}"] \arrow[r, phantom, "\leqslant"] & (K \rightharpoonup \mathsf{F}\,K) \times K \arrow[l, bend left, "\mathit{rec\text{-}intro}"] \end{tikzcd} \qquad (5.10)$$

for K a datatype of *"heap addresses"*, *keys* or *"pointers"*, such that $K \cong I\!N$. For example, the binary tree on the left-hand side of (5.11) below will be represented — via (5.10) followed by (5.4) — by address 5 pointing at the tables on the right-hand side:



K	K'	K''
5	3	4
3	1	2

K	$Leaf$
1	a
2	b
4	c

(5.11)

See [79, 80, 81] for several important details we have to skip at this point about this *generic* data representation technique, in particular in what concerns the complex abstraction invariant imposed by (5.10), which requires "well-founded heaps".

5.1.4 Normal form

A pattern common to equations (5.3, 5.4, 5.6, 5.7, 5.8, 5.9 and 5.10) is that right-hand-sides do not involve functors other than product ($\times$) and finite mapping ($\rightharpoonup$). It so happens that these are exactly the functors admissible in the following abstract model

$$DB = \prod_{i=1}^{n} \left(\prod_{j=0}^{n_i} K_j \rightharpoonup \prod_{k=0}^{m_i} D_k \right) \qquad (5.12)$$

of a relational database, whereby every $db \in DB$ is a collection of n relational tables (index $i = 1, n$) each of which is a mapping from a tuple of *keys* (index j) to a tuple of relevant *data* (index k). Wherever $m_i = 0$ we have $\prod_{k=0}^{0} D_k \cong 1$, meaning — via (5.6) — that we have a *finite set* of tuples in $\prod_{j=0}^{n_i} K_j$. (These are called *entity relationships* in the standard terminology.) Wherever $n_i = 0$ we are in presence of a singleton relational table. Last but not least, all K_j and D_k are assumed to be "atomic" types, otherwise db would fail first normal form (1NF) compliance [68].

To derive such normal forms, the above calculation laws can be used in combination with appropriate laws for commutativity and associativity of tuples, and laws for introduction and elimination of empty tuples. To avoid these additional bookkeeping laws, we can generalize law (5.9) to:

$$A \rightharpoonup (\textstyle\prod_i B_i \times \prod_j (C_j \rightharpoonup D_j)) \quad \underset{\xleftarrow[g\text{-}unnjoin]{\qquad\qquad}}{\overset{\xrightarrow{g\text{-}njoin}}{\leqslant}} \quad (A \rightharpoonup \textstyle\prod_i B_i) \times \prod_j (A \times C_j \rightharpoonup D_j) \tag{5.13}$$

In the implementation, we will make use of this generalization.

Thus, with this collection of calculation rules we are able to unravel (polynomial) recursive datatypes and decompose complex/nested mappings or sequences into tuples of simpler mappings, leading to models in relational normal form (5.12). In the upcoming section we will show how a term rewriting system can be constructed and implemented that performs such unraveling in a deterministic and confluent manner.

5.2 Design and Implementation

In this section we will explain how the bridge from this refinement theory to practice was done using strategic term rewriting [95, 22, 66].

The overall architecture of the tool, shown in Figure 5.1, mirrors the phases needed to tackle the problem:

1. Recognize a specification file written in VDM-SL and convert it to a format that can be used for processing: abstract syntax tree (AST);

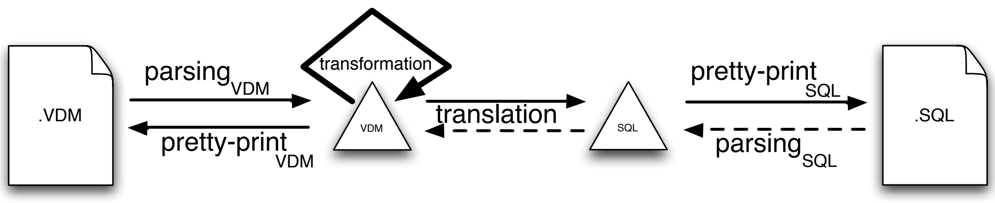


Figure 5.1: Overall architecture of the VooDooM *tool.*

2. Apply transformations to the AST to convert the input model into its relational equivalent; and
3. Output the transformed specification either as VDM-SL or to SQL concrete syntax.

To handle each of these steps, the following modules are necessary:

VDM-SL and SQL front-ends Deal with the language issues, namely parsing, pretty-printing and abstract representation.

Transformation engine Receives a VDM-SL AST representing the original specification and applies the calculation laws in order to compute a relational model that refines it (also a VDM-SL AST).

VDM-SL to SQL translator Maps a relational model in VDM-SL AST format to an equivalent SQL AST.

The VDM-SL front-end has already been introduced in Chapter 3 and the SQL front-end was provided by the Software Improvement Group which gave us permission to use.

In the upcoming sections we will describe the implementation of both the transformation engine and the VDM-SL to SQL translator.

We will use the following specification of a tiny bank account management system (BAMS) [5] as input example[1]:

[1]The intermediate steps will be presented in concrete VDM-SL syntax for clarity, although the tool actually uses ASTs.

types

7.0 $BAMS = AccId \xrightarrow{m} Account;$

8.0 $Account :: H : AccHolder$-set
.1 $B : Amount;$

9.0 $AccId =$ char*;

10.0 $AccHolder =$ char*;

11.0 $Amount = \mathbb{Z}$

5.2.1 Transformation

The transformation engine is the core module of the VooDooM tool. It is responsible for the refinement of the VDM-SL datatypes to a relational form, in accordance with the refinement laws presented above.

Its implementation makes ample use of strategic term rewriting techniques. The overall approach is as follows. First, we formulate individual term rewriting rules on the basis of the type signatures of the representation functions of the refinement laws. Table 5.1 lists these individual rules. Secondly, we use strategy combinators to compose these individual rules into a transformation engine that applies the individual rules in a way that a normal form is reached in a deterministic and confluent manner.

Before transformation begins, a single traversal is made over the AST that represents the complete VDM-SL input specification and collects all sub-ASTs that represent datatype definitions into a list. The transformation process itself operates on this collection and is organized into the following sequential phases:

Function	Rewrite rule	Law
seq2index	$A^\star \Rightarrow I\!\!N \rightharpoonup A$	(5.3)
unconjoin	$A \rightharpoonup (B + C) \Rightarrow (A \rightharpoonup B) \times (A \rightharpoonup C)$	(5.4)
distr	$A \times (B + C) \Rightarrow (A \times B) + (A \times C)$	(5.5)
set2fm	$2^A \Rightarrow A \rightharpoonup 1$	(5.6)
opt-elim	$A + 1 \Rightarrow 1 \rightharpoonup A$	(5.7)
unpeither	$(B + C) \rightharpoonup A \Rightarrow (B \rightharpoonup A) \times (C \rightharpoonup A)$	(5.8)
unnjoin	$A \rightharpoonup (B \times (C \rightharpoonup D)) \Rightarrow (A \rightharpoonup B) \times (A \times C \rightharpoonup D)$	(5.9)
rec-elim	$\mu\mathsf{F} \Rightarrow (K \rightharpoonup \mathsf{F}\,K) \times K$	(5.10)

Table 5.1: Catalog of rewriting rules. These rules are based on the various equations and inequations of the calculational data refinement theory presented in Section 5.1.

Inlining and recursion removal

The rewrite rules for conversion operate on datatypes, not on systems of named datatype definitions. To avoid needing to perform lookups of datatype names during transformation, we start by inlining, i.e. replacing all datatype names by their definitions. This technique leads to the loss of the top level datatype names, which in some cases are useful. To overcome this problem, singleton composes are introduced before inlining those types.

Of course, this substitution process would run into cycles if we did not treat recursive definitions differently. For this reason, the recursion removal rewrite rule *rec-elim* is used in combination with inlining. After these rules have exhaustively been applied, a set of non-recursive, independent datatypes is obtained that is amenable to further transformation. Exhaustive application is realized by using the *repeat* combinator

After the inlining step, our example specification will look as follows:

```
types

12.0  BAMS =
  .1      AccId ::  char* -m-> Account ::
  .2                    H : (AccHolder ::     char*)-set
  .3                    B : Amount ::     ℤ
```

Though our example does not contain recursive datatypes, the tree example of Section 5.1 illustrates recursion removal. More examples are given in [79, 80].

Desugaring

We limit the language of datatype definitions by removing those constructs for which we have a simple elimination rule: sets, sequences, and optionals. Sequences of characters are viewed as atomic and excluded from desugaring, because we want to map them to native SQL strings (varchar). Also, we rewrite all tuples to VDM-SL's `compose` construct. This desugaring step is performed by applying the rules *seq2index*, *set2fm*, and *optElim*, in a single traversal.

In the same traversal, we rewrite tuples to VDM-SL compose constructs. Alternatively, we could have desugared composes to nested tuples, but that would lead to the loss of names of composes and their fields. Of course, if all tuples are eliminated in favour of composes, this has the consequence that all calculation laws involving products should be mapped to rewrite rules involving composes. This has as additional benefit that various rules, e.g. (5.13), can be generalized, because composes are n-ary, rather than binary.

After desugaring, our example specification looks as follows:

```
types

13.0  BAMS =
  .1      AccId ::  char* -m-> Account ::
  .2                    H : (AccHolder ::     char*) -m-> NIL
  .3                    B : Amount ::     ℤ
```

This expression contains only maps and products (compose), but is not yet in relational form.

Conversion to relational form

Once the desugared structure is obtained, further transformation rules can now be applied. At this stage, the rules to apply are *unconjoin*, *unpeither*, and the generalized version of *unnjoin*. In addition, a rule for flattening nested composes is needed to bring expressions into the best form to be rewritten via that generalized rule. These rewrite rules need to be applied exhaustively throughout the AST. The *innermost* combinator is suitable for this.

After conversion, our example specification is in the relational normal form which follows:

```
types

14.0  BAMS =
  .1      compose mapAggr of
  .2          AccId ::  char* -m-> Amount ::  ℤ
  .3          tuple ::  char*  char* -m-> NIL
  .4      end
```

Resugaring

Finally, sets are reintroduced into the expression, using the `dom` rule (5.6). Thus, any occurrence of the form `map x to NIL` is converted to `set of x`. This occurs when further simplification is not possible. This is justified, because these can be represented directly in SQL. When VDM-SL is targeted as output language, tuples are reintroduced where binary composes with anonymous fields occur.

5.2.2 SQL Translation

During transformation, an initial specification is transformed into a relational normal form. In the translation process these VDM-SL datatypes are converted to SQL tables and attributes.

VDM-SL datatype	SQL datatype	SQL Constraint
bool	SMALLINT	CHECK (.. IN (0,1))
nat	INT	CHECK .. >= 0
nat1	INT	CHECK .. >= 1
int	INT	
rat	REAL	
real	REAL	
char	CHAR (1)	
token	VARCHAR (128)	
seq of char	VARCHAR (128)	

Table 5.2: Correspondence between VDM-SL and SQL92 datatypes.

The translation of normal forms to SQL is straightforward. The relational equivalent of a *map* is a table in which the domain of the map is the primary key. The relational counterpart of a *set* is a table with a compound primary key on all columns to guarantee uniqueness. The elements of maps and sets, which are products of elementary VDM-SL datatypes, are converted to SQL column attributes (which are also of elementary types).

Because basic VDM-SL and SQL datatypes are not compatible, a correspondence between them must be made. Table 5.2 shows the correspondence as implemented in the VooDooM tool, which also shows constraints to be added to the SQL data model to better preserve the semantics of some VDM-SL datatypes. Only Standard SQL92 [34] datatypes were chosen, thus providing a solution that works for all SQL vendor dialects.

The SQL generated for our running BAMS example is as follows:

```
CREATE TABLE table1 (                CREATE TABLE table2 (
 AccId VARCHAR (128) NOT NULL,        Attr1 VARCHAR (128) NOT NULL,
 Amount INT NOT NULL,                 Attr2 VARCHAR (128) NOT NULL,
 PRIMARY KEY (AccId)                  PRIMARY KEY (Attr1, Attr2)
)                                    )
```

As can be seen, a composite type (the outer compose) with a map and a set (reintroduced for `map ... to NIL`) is translated to two tables in SQL. Because none of the compose elements have tags, they have been automatically generated as `table1` and `table2`. The fields of the inner composes have been converted to SQL attribute columns. In case of the map there are two tags: `AccId` and `Amount`. This led to the creation of two attributes with those names. The primary key of the generated table is `AccID` because it represents the domain of the map. In case of the set there are no tags, so attribute names are automatically generated: `Attr1` and `Attr2`. These two attributes together form a compound primary key, because combined they represent the domain of the set.

Thus, `table1` uniquely associates an amount to the identifier of each account in the system, while `table2` relates accounts identifiers to account holders. These two tables implement the original specification in which account identifiers are mapped to accounts, and each account has a set of account holders and an amount. The actual retrieve function that witnesses the abstraction relation between the original VDM-SL specification and this pair of SQL tables is given in [5].

5.3 Summary

In this chapter we describe a relational calculator which, based on refinement by calculation, automatically derives from a formal specification an equivalent relational model which can be translated to SQL.

Theory is introduced by presenting the conversion laws describing the transformations that can be performed. By applying these rules, a certain normal form must be reached. This indicates that an equivalent relational model is successfully obtained and translated to SQL.

The link from theory to practice is described by explaining how the previously defined transformation laws were implemented in Haskell and Strafunski and how a formal specification can be translated to SQL.

We conclude with a test case analysis of a small bank account specification.

Chapter 6

Conclusions and future work

This chapter presents a brief discussion about the work done carried out in the project which has led to this dissertation. Section 6.1, provides an overview about the development method which was strongly based on testing. For the three main parts of the work (VDM-SL front-end, understanding, and re-engineering) we will present related work in Section 6.2, contributions in Section 6.3, and future work in Section 6.4.

Finally, the availability of the tool is discussed in Section 6.5.

6.1 Development Method

It would be ironic that a work about software engineering wouldn't follow strong software engineering practices. Hence, all steps of this work were accompanied by versioning, testing and automatic documentation generation.

Versioning was an important instrument to both development of the different software components and the dissertation itself. It allowed us not only to track changes but also it acted as form of documentation. In each revision, careful attention was paid to supplied comments which would remind us why certain changes were made.

Testing was applied using two different testing frameworks/tools.

For the development of the VDM-SL grammar (Chapter 3) the *parse-*

unit tool was used for both integration and unit testing. We started with integration tests, using full specifications, to check the completeness of the grammar. The observation of failure in some tests indicated that the grammar was not complete due to typos, and missing or incomplete grammar rules. Since the first version of the grammar was manually transcribed, integration tests proved to be very valuable in discovering most errors in the grammar. After all integration tests succeeded (with ambiguities) we incrementally developed unit tests. Each unit test tried to reproduce exactly one kind of ambiguity found and, for each of these, a new revision of the grammar correcting that problem was produced. Testing finished when all reported ambiguities were solved.

This testing mechanism was possible thanks to the technology used, generalized LR parsing (GLR), which also allowed us to develop a full near flawless grammar in short time.

For the implementation of the analysis and the VDM-SL re-engineering, Sections 4 and 5 respectively, we used a simple but powerful Haskell testing framework: *HUnit* [44]. This framework has an extensible API to define assertions which we enriched with a few functions specifically intended to test both conversion and translation.

As a rule, for testing a function, we specify the input and the expected output values. Test observations assert that the values produced from the input values are equal to the expected output values.

When specifying VDM-SL transformations, the inputs and the outputs of our tests were VDM-SL AST datatypes. Furthermore, when specifying VDM-SL to SQL translations, the inputs were VDM-SL AST datatypes and the expected outputs were SQL AST datatypes.

Every time we wanted to specify a test we had the lengthy and error prone task of browsing through the AST datatype and selecting the right types to use. This proved to be a burden, not only because it was difficult to figure out the correspondence between what we wanted to express and the correct datatypes but also because of the large number of datatypes envolved to expressing even small examples.

Because this was found to be impractical we decided to find a better alternative to consistently express the tests and, therefore, we decided to switch to concrete syntax for specifying both input values and expected results.

This was implemented by parsing code in concrete syntax and subsequent pass the parsing result to the functions we wanted to test. Both in transformation and translation this proved to be a success by allowing us to write faster and far more readable test cases.

Also, using concrete syntax has the additional benefit that even if we decide to change the AST our tests will resist to that change, i.e., we do not need to change our unit tests, and consequently, new errors can not be introduced.

Finally, for automatic documentation generation the *Haddock* tool [70] was used to generate API HTML documentation.

6.2 Related Work

In this section we will present a brief overview about the state of art of field in which our work is related with, namely grammar engineering, program understanding and re-engineering.

For each of these, we will devote a section to discuss how our work comparables to other author's work.

6.2.1 Front-end for VDM-SL

Malloy and Power have applied various software engineering techniques during the development of a LALR parser for C# [69]. Their techniques include versioning, testing, and the grammar size, complexity, and structure metrics that we adopted ([83], see Table 3.1). They do neither measure coverage, nor unit testing.

Lämmel et al. have advocated derivation of grammars from language reference documents through a semi-automatic transformational process [64, 62]. 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 do. No coverage measurement nor unit tests are reported.

Klint et al. provide a survey over grammar engineering techniques and an agenda for grammar engineering research [57]. The need for an engineering approach and tool support for grammar development has also been recognized in the area of natural language processing [29, 98].

6.2.2 Support for understanding

References [1, 42] report studies where formal specifications were manually developed from real-world systems, based on interviews, requirements documents or simulation. Better understanding about those systems was achieved due to the higher level of abstraction of the formal specifications.

Moreover, IFAD has implemented an automatic Java to VDM++ conversion in the VDMTools [49] to support reverse engineering of legacy Java applications to VDM++. Java class files are analyzed and equivalent VDM++ specifications are produced. VDMTools allow for two kinds of specification generation: stubs only in which only class attributes and functions signature are defined; and Java code in which automatic transformations to code are applied.

Both manual and automatic processes provide better understanding of existent systems by raising the abstraction level of the language.

6.2.3 Support for re-engineering

Most work on formal methods in relational database design is concerned with formal models of relational data. This interest dates back to (at least) [14], where a formalization of a relational data model is given using the VDM notation.

The formal specification and design of a program implementing simple update operations on a binary relational database called NDB is described in [100]. This single level description of NDB is the starting point of [31], where a case study in the modular structuring of this "flat" specification is presented. The authors present a second specification which makes use of an n-ary relation module, and a third one which uses an n-ary relation module with type and normalization constraints. They demonstrate the reusability of their modules, and also outline specifications of an n-ary relational database with normalization constraints, and an n-ary relational database with a two-level type hierarchy and no normalization constraints. However, their emphasis is on the modularization techniques adopted to organize VDM specifications into modules.

Samson and Wakelin [86] present a comprehensive survey about the use of algebraic methods to specify databases. They compare a number of approaches according to the features covered and enumerate some features not normally covered by such methods.

Barros [12] describes an extension to the traditional database design aimed at formalizing the development of (relational) database applications. A general method for the specification of relational database applications using Z [87] is presented. A prototype is built to support the method. It provides for editing facilities and is targeted at the DBPL database management system.

The purpose of Baluta [11] is to rigorously specify the basic features of the relational data model version 2 (RM/V2) as defined by Codd [25], using the Z language.

More recently, Necco [74] exploits aspects of *data processing* which are functional in nature and can take advantage of recent developments in the area of *generic functional programming* and calculi. Generic Haskell [45, 23] is used to animate a generic model of a subset of the standard relational database calculus, written in the style of model-oriented formal specification.

6.3 Contributions

6.3.1 Front-end for VDM-SL

In accordance with the objectives set in Section 3, the contributions of the work reported in this dissertation are twofold. On the one hand, we contribute to the body of grammar engineering knowledge. We show how grammar testing, grammar metrication, and coverage analysis can be combined in a systematic grammar development process that can be characterised as a tool-based methodology for iterative grammar development. We have motivated the use of these grammar engineering techniques from the larger context of grammar-centered language tool development, and we have conveyed our experiences in using them in a specific grammar development project. In the process, we have extended the collection of metric reference data of [83] with values for 6 additional grammars of widely used languages, and we have collected data for additional grammar metrics defined by us in [8].

On the other hand, we document the developed VDM-SL grammar itself as well as the iterative process that led to its creation. We have monitored the development process with various grammar metrics, and developed unit tests to guide the process of its disambiguation and refactorization. The presented metrics values quantify the size and complexity of the grammar and reveal the level of continuity during evolution. The documented process establishes a firm link between the ISO standard (that served as starting point) and the final, delivered grammar.

Although this grammar was specially developed for this work, we hope it will be useful to formal method tool initiatives such as the Overture project[1].

6.3.2 Support for understanding

In contrast with other projects where understanding was achieved by converting real-world systems knowledge to formal specifications, our primary

[1]See `http://www.overturetool.org/`.

contribution was the application of known program understanding techniques to extract yet another kind of information from formal specifications.

We have looked to formal specifications as ordinary software artifacts suffering of all maintainability problematic as discussed in [67] despite their higher level of abstraction.

Additionally, we developed a framework for formal specifications understanding with the functionalities to extract procedure call, type dependency and procedure-type dependency graphs which, subsequently, were used as input of program understanding techniques.

Strongly connected components analysis and formal concept analysis as well as software metrics were introduced to another software engineering field, formal methods, with the hope they can be used for instance in re-engineering or other formal methods tools.

6.3.3 Support for re-engineering

Twelve years ago, Barros [12] referred to the derivation of database programs directly from formal specifications as an unsolved problem. By contrast, deriving the database structure was regarded as a trivial aspect. However, his specifications are Z schemata whose internal states are already close to the relational model (e.g. power-sets of products).

This is in contrast with our approach, in which the source data-model can be arbitrarily complex (as far as VDM-SL data constructors are concerned), including recursive datatypes. Our "derecursivation" law (*rec-elim*), which relationally expresses the main result of [99], bears some resemblance (at least in spirit) with "defunctionalization" [48], a technique which is used in program transformation and compilation.

On the other hand, our approach shares with [12] the view that database design should be regarded as special case of data refinement. It is orthogonal to [12] in the sense that we are not concerned with database dynamics (transactions, etc).

Another advantage of our approach is the prospect of synthesizing abstraction invariants generated by each refinement step, which is still in the *to-do* list of the project. These include abstraction or representation functions and concrete invariants. The former can be used for *data-migration* between the original VDM-SL source and the generated relational model, in a way similar to [76] and to what is done manually in [5]. Recently, this issue was covered in [27]. The latter can be (at least in part) incorporated as SQL constraints.

Strategic term rewriting provides a realistic solution to database schema calculation when compared with previous attempts to animate the same calculus using *genetic algorithm*-based term-rewriting techniques [75].

6.4 Concluding remarks and future work

We successfully achieved all the objectives set in Section 1.4. However, there is still space for improvement.

6.4.1 Front-end for VDM-SL

Lämmel [60] generalized the notion of rule coverage and advocates the uses of coverage analysis in grammar development. 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++. VDM++ has the advantage of the Java to VDM++ converting supported by the VDMTools which would allow us to have access to a larger specification test cases.

We have already generated Haskell support for VDM processing for the grammar, and are planning to provide Java support as well. This could enable other projects to develop tool support for formal methods.

Also, the integration test suite deserves further extension in order to increase coverage.

6.4.2 Support for understanding

A basic framework for program understanding was built and, with it, different kinds of graphs and metrics were produced.

We briefly showed how to interpret these graphs and commented on some metric results. Within some limitations, both extracted and derived graphs were successfully used for interpretation. We realized, for instance, that the results of transitive closure in the presence of many cycles is not useful, as well as the strongly connected components one, which does not add much information when no cycles exist.

Since interpretation of the analysis results was not the main focus of the work, this was not done extensively.

We showed how we could answer specific questions using some kinds of graphs and to discover more information when combining different kinds of graphs. The process for discovering information was done in a more or less informal way. This could be improved by creating, for instance, a catalog of procedures to follow when answering specific questions. Also, only the metric values of a single graph of the SAFER specification were reported and commented. To use metrics results in statistical analysis we must apply them to a more comprehensive set of systems.

In summary, as future work we should focus on interpretation and study in which scenarions such analyses could be applied.

Another issue that could be worth of research is the applicability of these "research data" in re-engineering, aiding transformation or translation. This would provide a closer link between understanding and re-engineering.

Finally, we could improve our implementation of techniques, such as eg. the procedure-type flow analysis, and extend these with others known from the literature [77].

A possible improvement in procedure-type flow analysis is to take into account the state of a specification. This was not done because, in ISO VDM-SL, only one state definition can be specified and we assumed that only one file can be analyzed. If IFAD VDM-SL extensions were used, which support modules and a state definition per module, the improvement of this analysis becomes mandatory to enable to identify which operations manipulate which state.

Moreover, as possible extensions, we could consider techniques that take into account the body of functions and operations.

Furthermore, we only applied our analysis to a single test case. To validate these techniques it is necessary to test different methodologies for applying these analysis during development and maintenance and to apply them to a significant number of specifications.

6.4.3 Support for re-engineering

We would like to extend the re-engineering process in several ways. Firstly, in addition to the conversion of VDM-SL to SQL, we want to support the reverse process of obtaining an algebraic set of datatypes from a relational model, as already suggested by the dashed lines in the architecture overview in Figure 5.1.

Reversing a database to VDM-SL is not a novelty. This problem was already tackled in [76], in which the authors describe an implemented functional prototype and its application using a real world example.

However that implementation has several drawbacks. The process has to be assisted manually, the initial relational model must be specified in VDM-SL, the transformation rules were coded with explicit recursion, and all traversals were hard-coded leading to inflexibility in the implementation.

With strategic term rewriting approach, the same problem can be solved in a more pragmatic way.

Secondly, we intend to offer better support for invariants to the tool. The transformation and translation processes in both directions lack support for VDM-SL invariants. To more accurately preserve semantics, invariants should be added during the transformation process when a datatype is split in two or more.

However, invariants pose some difficulties when performing transformations since the data definitions which they refer to are changing. Thus, invariants need also to reflect this change. When the transformations are simple, rearrangements of data fields can be easy but, since invariants can be as elaborate as any function mapping the type to a boolean, the general case is not. Transforming arbitrary functionality in an automated manner is a challenging subject which would involve research beyond the scope of this tool. However, we intend to develop some invariant support, namely to referential integrity constraints, by providing a small subset of VDM-SL that can be mapped into SQL constraints in an automated way.

6.5 Availability

The VooDooM tool presented in this work is freely available under the BSD open-source license from `http://voodoom.sourceforge.net/`.

From the website it is possible to download both the source code and binary versions of the VooDooM tool for several operating systems. To ease the process, third-party software dependencies are also available for download.

In addition to the tool, the ISO VDM-SL grammar in SDF was also made available as browseable hyperdocument.

Bibliography

[1] S. Agerholm and P. G. Larsen. Modeling and validating SAFER in VDM-SL. In M. Holloway, editor, *Fourth NASA Langley Formal Methods Workshop*, September 1997.

[2] S. Agerholm and P. G. Larsen. SAFER specification in VDM-SL, 1997.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] A. Albrecht. Measuring application development productivity. In I. B. M. Press, editor, *IBM Application Development Symp.*, pages 83–92, October 1979.

[5] J. Almeida, L. Barbosa, F. Neves, and J. Oliveira. Bringing Camila and SetCalc Together — the `bams.cam` and `ppd.cam` Camila Toolset demos. Technical report, DI/UM, Braga, December 1997. [45 p. doc.].

[6] T. L. Alves, P. F. Silva, J. Visser, and J. N. Oliveira. Strategic term rewriting and its application to a vdm-sl to sql conversion. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 2005.

[7] T. L. Alves and J. Visser. Grammar engineering applied for development of a VDM grammar. Submitted for publication, 2005.

[8] T. L. Alves and J. Visser. Metrication of SDF grammars. Technical Report DI-PURe-05.05.01, Universidade do Minho, May 2005.

[9] K. Arnold and J. Gosling. *The Java programming language*. Addison-Wesley, 1996. ISBN 0-201-63455-4.

[10] R. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T. Voermans, and J. van der Woude. Polynomial relators. In *2nd Int. Conf. Algebraic Methodology and Software Technology (AMAST'91)*, pages 303–362. Springer LNCS, 1992.

[11] D. D. Baluta. A formal specification in Z of the relational data model, Version 2, of E.F. Codd. M. Sc. thesis, Concordia University, Montreal, QC, Canada, 1995.

[12] R. Barros. Deriving relational database programs from formal specifications. In M. Naftalin, B. T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 24-18, 1994, Proceedings*, volume 873 of *Lecture Notes in Computer Science*, pages 703–723. Springer, 1994.

[13] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.

[14] D. Bjorner and C. Jones. *Formal Specification and Software Development*. Series in Computer Science. Prentice-Hall International, 1982. C.A.R. Hoare, ed.

[15] P. E. Black. directed graph. Dictionary of Algorithms and Data Structures. http://www.nist.gov/dads/HTML/directedGraph.html.

[16] M. v. d. Brand, A. v. Deursen, J. Heering, H. d. Jonge, M. d. 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.

[17] M. v. d. 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.

[18] M. Bravenboer. Parse Unit home page. `http://www.program-transformation.org/Tools/ParseUnit`.

[19] F. Buchli. Detecting software patterns using formal concept analysis. Diploma thesis, University of Bern, September 2003.

[20] B. Cheng and G. Gannod. A two-phase approach to reverse engineering using formal methods. In *Formal Methods in Programming and Their Applications*. Springer-Verlag, 1993. Lecture Notes in Computer Science.

[21] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[22] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Foundations of Software Science and Computation Structures, ETAPS'2001*, Lecture Notes in Computer Science, pages 166–180, Genova, Italy, Apr. 2001. Springer-Verlag.

[23] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The generic Haskell user's guide, November 2001. Technical Report UU-CS-2001-26, Universiteit Utrecht.

[24] E. Codd. A relational model of data for large shared data banks. *Communications of ACM*, 13(6):377–387, June 1970.

[25] E. F. Codd. *Missing Information.* Addison-Wesley Publishing Company, Inc., 1990.

[26] A. Cruz. Objectificao de especificaes formais. Master's thesis, University of Minho, 2004.

[27] A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation. In *Formal Methods'06, Lecture Notes in Computer Science*, jul 2006.

[28] 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.

[29] G. Erbach. Tools for grammar engineering, Mar. 15 2000.

[30] N. Fenton and S. Pfleeger. *Software metrics: a rigorous and practical approach.* PWS Publishing Co., Boston, MA, USA, 1997. 2nd edition, revised printing.

[31] J. Fitzgerald and C. Jones. *Modularizing the formal description of a database system*, volume 428 of *Lecture Notes in Computer Science.* Springer, 1990.

[32] J. Fitzgerald and P. G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, United Kingdom, 1998.

[33] K. F. Fogel. *Open Source Development with CVS.* Coriolis Group Books, 1999.

[34] I. O. for Standardization. *Information Technology – Database languages – SQL.* Reference number ISO/IEC 9075:1992(E), Nov. 1992.

[35] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[36] B. Ganter and R. Wille. Applied lattice theory: Formal concept analysis, 1997.

[37] B. Ganter, . Willie, and C. Franzke. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1st edition, December 1998.

[38] H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002. ISBN: 0-13-031995-3.

[39] T. V. T. Group. The IFAD VDM++ language. Technical report, IFAD, Oct. 2000. ftp://ftp.ifad.dk /pub/vdmtools/doc/langmanpp_letter.pdf.

[40] T. V. T. Group. VDM-SL toolbox user manual. Technical report, IFAD, October 2000.

[41] M. H. Halstead. *Elements of software science (Operating and programming systems series)*. Elsevier, 1977.

[42] K. M. Hansen. Modelling Railway Interlocking Systems. Technical Report ID-TR: 1996-167, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, September 1995.

[43] 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.

[44] D. Herrington. Hunit user's guide 1.0., 2002.

[45] R. Hinze and J.Jeuring. Generic Haskell: Practice and theory, 2003.

[46] R. Holt. Binary relational algebra applied to software architecture, 1996.

[47] P. Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, University of Eindhoven, The Netherlands, 1997.

[48] G. Hutton and J. Wright. Compiling exceptions correctly. In D. Kozen and C. Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2004.

[49] IFAD, Forskerparken 10A, DK - 5230 Odense M. *The Java to VDM++ User Manual*, 2001.

[50] International Organisation for Standardization. *Information technology—Programming languages, their environments and system software interfaces—Vienna Development Method—Specification Language—Part 1: Base language*, Dec. 1996. ISO/IEC 13817-1.

[51] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[52] C. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C.A. R. Hoare.

[53] C. B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[54] S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.

[55] M. d. Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[56] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 2nd edition, 1988. ISBN 0-13-110362-8.

[57] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *Transaction on Software Engineering and Methodology*, 2005. To appear.

[58] V. Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Softw.*, 21(4):70–77, 2004.

[59] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. v. d. 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).

[60] R. Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[61] R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003.

[62] 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)*, Oct. 2004.

[63] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[64] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[65] R. Lämmel and J. Visser. Strategic polymorphism requires just two combinators! Technical Report cs.PL/0212048, arXiv, Dec. 2002.

[66] 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, Jan. 2003.

[67] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[68] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[69] 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.

[70] S. Marlow. Haddock, a haskell documentation tool. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 78–89, New York, NY, USA, 2002. ACM Press.

[71] T. J. Mccabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[72] E. E. Mills. *Software Metrics.* Software Engineering Institute - Carnegie Mellon University, 1998.

[73] C. Morgan. *Programming from Specification.* Series in Computer Science. Prentice-Hall International, 1990. C.A. R. Hoare, series editor.

[74] C. Necco. Polytypic data processing. Master's thesis, Facultad de Cs. Físico Matemáticas y Naturales, University of San Luis, Argentina, 2005. (Submitted.).

[75] F. Neves and J. Oliveira. ART — Um Laboratrio de Reificao "Gentica". In *IBERAMIA'98 — Sixth Ibero-Conference on Artificial Intelligence*, pages 201–215, Lisbon, Portugal, October 5-9 1998. (in Portuguese).

[76] F. Neves, J. Silva, and J. Oliveira. Converting Informal Meta-data to VDM-SL: A Reverse Calculation Approach . In *VDM in Practice! A Workshop co-located with FM'99: The World Congress on Formal Methods, Toulouse, France*, September 1999.

[77] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[78] J. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, April 1990.

[79] J. Oliveira. Software reification using the SETS calculus. In T. Denvir, C. B. Jones, and R. C. Shaw, editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. ISBN 0387197524, Springer-Verlag, 8–10 January 1992. (Invited paper).

[80] J. Oliveira. Data processing by calculation, 2001. 108 pages. Lecture Notes for the 6th Estonian Winter School in Computer Science, 4-9 March 2001, Palmse, Estonia.

[81] J. Oliveira. Calculate databases with 'simplicity', September 2004. Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK.

[82] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, 1995.

[83] J. Power and B. Malloy. A metrics suite for grammar-based software. In *Journal of Software Maintenance and Evolution*, volume 16, pages 405–426. Wiley, Nov. 2004.

[84] P. Purdom. Erratum: "A Sentence Generator for Testing Parsers" [BIT **12**(3), 1972, p. 372]. *BIT*, 12(4):595–595, 1972.

[85] S. Rugaber. Program comprehension, 1995.

[86] W. Samson and A. Wakelin. *Algebraic Specification of Databases: A Survey from a Database Perspective.* Workshops in Computing. Springer Verlag, Glasgow, 1992.

[87] J. Spivey. *The Z Notation — A Reference Manual.* Series in Computer Science. Prentice-Hall International, 1989. C.A. R. Hoare.

[88] I. H. T. Oetiker, H. Partl and E. Schlegl. *The Not So Short Introduction to LaTeX2e*, September 2005.

[89] S. Thompson. *Haskell- The Craft of Functional Programming.* Addison-Wesley, 1st edition, 1996. ISBN 0-201-40357-9.

[90] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[91] J. D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.

[92] M. 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.

[93] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 246–255, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[94] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.

[95] 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)*, Pont-à-Mousson, France, September 1998. Elsevier Science.

[96] 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).

[97] J. Visser. Structure metrics for XML schema. In *Proceedings of XATA 2006*, 2006.

[98] 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.

[99] E. G. Wagner. All recursive types defined using products and sums can be implemented using pointers. In C. Bergman, R. D. Maddux, and D. Pigozzi, editors, *Algebraic Logic and Universal Algebra in Computer Science*, volume 425 of *Lecture Notes in Computer Science.* Springer, 1990.

[100] A. Walshe. *NDB: The Formal Specification and Rigorous Design of a Single-User Database System.* Prentice Hall, ISBN 0-13-116088-5, 1990.

[101] E. W. Weisstein. Lattice theory. MathWorld–A Wolfram Web Resource.

[102] E. W. Weisstein. Strongly connected componen. MathWorld–A Wolfram Web Resource.

[103] E. W. Weisstein. Transitive closure. MathWorld–A Wolfram Web Resource.

[104] K. D. Welker and P. W. Oman. Software maintainability metrics models in practice. *Crosstalk, Journal of Defense Software Engineering*, 8(11):19–23, November/December 1995.

[105] R. Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. In *Rival, I. (ed) Ordered Sets*, pages 445–470, Dordrecht-Boston, Reidel, 1982.

Appendix A

SAFER specification

This appendix presents the original SAFER specification. The one used in our work is slightly different to conform to the ISO standard [50].

A.1 SAFER module

The top level state machine model of the controller is presented in the *SAFER* module. A transition function describes the effects of the controller's actions during a single frame. A 5 msec frame period is assumed (200 Hz sampling rate).

```
module SAFER
        imports
15.0      from TS all ,
16.0      from AAH all ,
17.0      from AUX all ,
18.0      from HCM all
        exports all
definitions
```

The top level state machine transition function represents one frame of controller operation (once around the main control loop). A post-condition is used to express some desired general properties (maximum 4 thrusters simultaneously and no two selected thrusters should oppose each other).

```
19.0  state SAFER of
  .1    clock : $\mathbb{N}$
```

```
.2      init $s \triangleq s = $ mk-$SAFER\,(0)$
.3    end
```

operations

```
20.0  $ControlCycle : HCM`SwitchPositions \times HCM`HandGripPosition \times$
.1                    $AUX`RotCommand \overset{o}{\to} TS`ThrusterSet$
.2    $ControlCycle\,($mk-$HCM`SwitchPositions\,(mode, aah), raw\text{-}grip, aah\text{-}cmd) \triangleq$
.3      let $grip\text{-}cmd = HCM`GripCommand\,(raw\text{-}grip, mode),$
.4          $thrusters = TS`SelectedThrusters\,(grip\text{-}cmd, aah\text{-}cmd,$
.5                                                 $AAH`ActiveAxes\,(),$
.6                                                 $AAH`IgnoreHcm\,())$ in
.7      $(AAH`Transition(aah, grip\text{-}cmd, clock)\,;$
.8       $clock := clock + 1;$
.9       return $thrusters$ )
.10   post card $RESULT \leqslant 4 \wedge$
.11         $ThrusterConsistency\,(RESULT)$
```

functions

```
21.0  $ThrusterConsistency : TS`ThrusterName$-set $\to \mathbb{B}$
.1    $ThrusterConsistency\,(thrusters) \triangleq$
.2      $\neg\,(2^{\langle B1\rangle,\langle F1\rangle} \subseteq thrusters) \wedge$
.3      $\neg\,(2^{\langle B2\rangle,\langle F2\rangle} \subseteq thrusters) \wedge$
.4      $\neg\,(2^{\langle B3\rangle,\langle F3\rangle} \subseteq thrusters) \wedge$
.5      $\neg\,(2^{\langle B4\rangle,\langle F4\rangle} \subseteq thrusters) \wedge$
.6      $\neg\,(thrusters \cap 2^{\langle L1R\rangle,\langle L1F\rangle} \neq 2 \wedge thrusters \cap 2^{\langle R2R\rangle,\langle R2F\rangle} \neq 2) \wedge$
.7      $\neg\,(thrusters \cap 2^{\langle L3R\rangle,\langle L3F\rangle} \neq 2 \wedge thrusters \cap 2^{\langle R4R\rangle,\langle R4F\rangle} \neq 2) \wedge$
.8      $\neg\,(thrusters \cap 2^{\langle D1R\rangle,\langle D1F\rangle} \neq 2 \wedge thrusters \cap 2^{\langle U3R\rangle,\langle U3F\rangle} \neq 2) \wedge$
.9      $\neg\,(thrusters \cap 2^{\langle D2R\rangle,\langle D2F\rangle} \neq 2 \wedge thrusters \cap 2^{\langle U4R\rangle,\langle U4F\rangle} \neq 2)$
```

end *SAFER*

A.2 AUX module - Auxiliary defintions

In the *AUX* module various auxiliary value and type definitions for the SAFER example are provided.

```
module $AUX$
      exports all
definitions
values
```

22.0 $\mathit{arbitrary\text{-}value} = \textsf{mk-token}\,(1001);$

23.0 $\mathit{axis\text{-}command\text{-}set} : \mathit{AxisCommand}\textsf{-set} = 2^{\langle \mathit{Neg}\rangle, \langle \mathit{Zero}\rangle, \langle \mathit{Pos}\rangle};$

24.0 $\mathit{tran\text{-}axis\text{-}set} : \mathit{TranAxis}\textsf{-set} = 2^{\langle X\rangle, \langle Y\rangle, \langle Z\rangle};$

25.0 $\mathit{rot\text{-}axis\text{-}set} : \mathit{RotAxis}\textsf{-set} = 2^{\langle \mathit{Roll}\rangle, \langle \mathit{Pitch}\rangle, \langle \mathit{Yaw}\rangle};$

26.0 $\mathit{null\text{-}tran\text{-}command} : \mathit{TranCommand} = \{a \mapsto \langle \mathit{Zero}\rangle \mid a \in \mathit{tran\text{-}axis\text{-}set}\};$

27.0 $\mathit{null\text{-}rot\text{-}command} : \mathit{RotCommand} = \{a \mapsto \langle \mathit{Zero}\rangle \mid a \in \mathit{rot\text{-}axis\text{-}set}\};$

28.0 $\mathit{null\text{-}six\text{-}dof} : \mathit{SixDofCommand} = \textsf{mk-}\mathit{SixDofCommand}\,(\mathit{null\text{-}tran\text{-}command},$
.1 $\mathit{null\text{-}rot\text{-}command})$

types

29.0 $\mathit{AxisCommand} = \langle \mathit{Neg}\rangle \mid \langle \mathit{Zero}\rangle \mid \langle \mathit{Pos}\rangle;$

30.0 $\mathit{TranAxis} = \langle X\rangle \mid \langle Y\rangle \mid \langle Z\rangle;$

31.0 $\mathit{RotAxis} = \langle \mathit{Roll}\rangle \mid \langle \mathit{Pitch}\rangle \mid \langle \mathit{Yaw}\rangle;$

32.0 $\mathit{TranCommand} = \mathit{TranAxis} \xrightarrow{m} \mathit{AxisCommand}$

.1 $\textsf{inv}\ \mathit{cmd} \triangleq \textsf{dom}\ \mathit{cmd} = \mathit{tran\text{-}axis\text{-}set};$

33.0 $\mathit{RotCommand} = \mathit{RotAxis} \xrightarrow{m} \mathit{AxisCommand}$

.1 $\textsf{inv}\ \mathit{cmd} \triangleq \textsf{dom}\ \mathit{cmd} = \mathit{rot\text{-}axis\text{-}set};$

34.0 $\mathit{SixDofCommand} :: \mathit{tran} : \mathit{TranCommand}$
.1 $\mathit{rot} : \mathit{RotCommand}$

end AUX

A.3 HCM module - Hand controller module

The hand controller module (HCM) consists of a set of switches, a hand grip controller with integral pushbutton, and a set of crew displays. A six degree-of-freedom command is derived from four-axis hand grip inputs and the control mode switch position. It is assumed that switch debouncing is provided by a low-level hardware or software mechanism so that switch transitions in this model may be considered clean.

module HCM

imports

35.0 from AUX all

exports all

definitions
types

36.0 $SwitchPositions :: mode : ControlModeSwitch$
.1 $aah : ControlButton;$

37.0 $ControlModeSwitch = \langle Rot \rangle \mid \langle Tran \rangle;$

38.0 $ControlButton = \langle Up \rangle \mid \langle Down \rangle;$

39.0 $HandGripPosition :: vert : AUX`AxisCommand$
.1 $horiz : AUX`AxisCommand$
.2 $trans : AUX`AxisCommand$
.3 $twist : AUX`AxisCommand$

The hand grip provides four axes for command input, which are multiplexed by the control mode switch into the required six axes. The four degrees of freedom on the controller are vertical, horizontal, transverse (along the shaft), and twist (about the shaft).

functions

```
40.0  $GripCommand : HandGripPosition \times ControlModeSwitch \to AUX`SixDofCommand$
  .1  $GripCommand$ (mk-$HandGripPosition$ ($vert, horiz, trans, twist$), $mode$) $\triangleq$
  .2    let $tran = \langle X \rangle \mapsto horiz,$
  .3                $\langle Y \rangle \mapsto$ if $mode = \langle Tran \rangle$
  .4                then $trans$
  .5                else $\langle Zero \rangle,$
  .6                $\langle Z \rangle \mapsto$ if $mode = \langle Tran \rangle$
  .7                then $vert$
  .8                else $\langle Zero \rangle \rightharpoonup,$
  .9        $rot = \langle Roll \rangle \mapsto$ if $mode = \langle Rot \rangle$
  .10             then $vert$
  .11             else $\langle Zero \rangle,$
  .12             $\langle Pitch \rangle \mapsto twist,$
  .13             $\langle Yaw \rangle \mapsto$ if $mode = \langle Rot \rangle$
  .14             then $trans$
  .15             else $\langle Zero \rangle \rightharpoonup$ in
  .16   mk-$AUX`SixDofCommand$ ($tran, rot$)
```

end *HCM*

A.4 TS module - Thruster selection logic

The truster selection logic of modelled in the *TS* module. Hand controller and AAH commands are merged together in accordance with tha various priority rules, yielding a six degree-of-freedom command. Thruster selection tables are consulted to convert translational and rotational components to individual actuator commands for opening suitable thruster valves.

```
module $TS$
      imports
41.0    from $AAH$ all ,
42.0    from $AUX$ all

      exports all
definitions
types

43.0  $ThrusterName = \langle B1 \rangle \mid \langle B2 \rangle \mid \langle B3 \rangle \mid \langle B4 \rangle \mid \langle F1 \rangle \mid \langle F2 \rangle \mid \langle F3 \rangle \mid \langle F4 \rangle \mid$
  .1                 $\langle L1R \rangle \mid \langle L1F \rangle \mid \langle R2R \rangle \mid \langle R2F \rangle \mid \langle L3R \rangle \mid \langle L3F \rangle \mid \langle R4R \rangle \mid$
  .2                 $\langle R4F \rangle \mid \langle D1R \rangle \mid \langle D1F \rangle \mid \langle D2R \rangle \mid \langle D2F \rangle \mid \langle U3R \rangle \mid$
  .3                 $\langle U3F \rangle \mid \langle U4R \rangle \mid \langle U4F \rangle;$

44.0  $ThrusterSet = ThrusterName$-set

functions
```

```
45.0  $RotCmdsPresent : AUX‘RotCommand \to \mathbb{B}$

  .1  $RotCmdsPresent\,(cmd) \triangleq$
  .2     $\exists\, a \in \mathsf{dom}\ cmd \cdot cmd\,(a) \neq \langle Zero \rangle$;
```

```
46.0  $PrioritizedTranCmd : AUX‘TranCommand \to AUX‘TranCommand$

  .1  $PrioritizedTranCmd\,(tran) \triangleq$
  .2     if $tran\,(\langle X \rangle) \neq \langle Zero \rangle$
  .3     then $AUX‘null\text{-}tran\text{-}command \dagger \langle X \rangle \mapsto tran\,(\langle X \rangle) \rightharpoonup$
  .4     elseif $tran\,(\langle Y \rangle) \neq \langle Zero \rangle$
  .5     then $AUX‘null\text{-}tran\text{-}command \dagger \langle Y \rangle \mapsto tran\,(\langle Y \rangle) \rightharpoonup$
  .6     elseif $tran\,(\langle Z \rangle) \neq \langle Zero \rangle$
  .7     then $AUX‘null\text{-}tran\text{-}command \dagger \langle Z \rangle \mapsto tran\,(\langle Z \rangle) \rightharpoonup$
  .8     else $AUX‘null\text{-}tran\text{-}command$;
```

```
47.0  $CombinedRotCmds : AUX‘RotCommand \times AUX‘RotCommand \times$
  .1                        $AUX‘RotAxis\text{-}\mathsf{set} \to AUX‘RotCommand$

  .2  $CombinedRotCmds\,(hcm\text{-}rot, aah, ignore\text{-}hcm) \triangleq$
  .3     let $aah\text{-}axes = ignore\text{-}hcm \cup$
  .4                 $\{a \mid a \in AUX‘rot\text{-}axis\text{-}set \cdot hcm\text{-}rot\,(a) = \langle Zero \rangle\}$ in
  .5     $\{a \mapsto aah\,(a) \mid a \in aah\text{-}axes\} \overset{m}{\cup}$
  .6     $\{a \mapsto hcm\text{-}rot\,(a) \mid a \in AUX‘rot\text{-}axis\text{-}set \setminus aah\text{-}axes\}$;
```

```
48.0  $IntegratedCommands : AUX‘SixDofCommand \times AUX‘RotCommand \times$
  .1                        $AUX‘RotAxis\text{-}\mathsf{set} \times AUX‘RotAxis\text{-}\mathsf{set} \to AUX‘SixDofCommand$

  .2  $IntegratedCommands\,($mk-$AUX‘SixDofCommand\,(tran, rot),$
  .3                        $aah, active\text{-}axes, ignore\text{-}hcm) \triangleq$
  .4     if $AAH‘AllAxesOff\,(active\text{-}axes)$
  .5     then if $RotCmdsPresent\,(rot)$
  .6           then mk-$AUX‘SixDofCommand\,(AUX‘null\text{-}tran\text{-}command, rot)$
  .7           else mk-$AUX‘SixDofCommand\,(PrioritizedTranCmd\,(tran),$
  .8                                        $AUX‘null\text{-}rot\text{-}command)$
  .9     else if $RotCmdsPresent\,(rot)$
  .10         then mk-$AUX‘SixDofCommand\,(AUX‘null\text{-}tran\text{-}command,$
  .11                                       $CombinedRotCmds\,(rot, aah, ignore\text{-}hcm))$
  .12         else mk-$AUX‘SixDofCommand\,(PrioritizedTranCmd\,(tran), aah)$;
```

Selection of back and forward thrusters results in a pair of thruster sets, the first of which gives mandatory thrusters and the second gives optional thrusters. (Mandatory means always selected while optional means conditionally selected.) This function represents the selection table for X, pitch, and yaw commands.

```
49.0  $BFThrusters : AUX`AxisCommand \times AUX`AxisCommand \times$
 .1                  $AUX`AxisCommand \rightarrow ThrusterSet \times ThrusterSet$
 .2   $BFThrusters\,(A, B, C) \triangleq$
 .3     cases mk- $(A, B, C)$ :
 .4       mk- $(\langle Neg\rangle, \langle Neg\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle B4\rangle}, 2^{\langle B2\rangle,\langle B3\rangle})$,
 .5       mk- $(\langle Neg\rangle, \langle Neg\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle B3\rangle,\langle B4\rangle}, 2)$,
 .6       mk- $(\langle Neg\rangle, \langle Neg\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle B3\rangle}, 2^{\langle B1\rangle,\langle B4\rangle})$,
 .7       mk- $(\langle Neg\rangle, \langle Zero\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle B2\rangle,\langle B4\rangle}, 2)$,
 .8       mk- $(\langle Neg\rangle, \langle Zero\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle B1\rangle,\langle B4\rangle}, 2^{\langle B2\rangle,\langle B3\rangle})$,
 .9       mk- $(\langle Neg\rangle, \langle Zero\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle B1\rangle,\langle B3\rangle}, 2)$,
 .10      mk- $(\langle Neg\rangle, \langle Pos\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle B2\rangle}, 2^{\langle B1\rangle,\langle B4\rangle})$,
 .11      mk- $(\langle Neg\rangle, \langle Pos\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle B1\rangle,\langle B2\rangle}, 2)$,
 .12      mk- $(\langle Neg\rangle, \langle Pos\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle B1\rangle}, 2^{\langle B2\rangle,\langle B3\rangle})$,
 .13      mk- $(\langle Zero\rangle, \langle Neg\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle B4\rangle,\langle F1\rangle}, 2)$,
 .14      mk- $(\langle Zero\rangle, \langle Neg\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle B4\rangle,\langle F2\rangle}, 2)$,
 .15      mk- $(\langle Zero\rangle, \langle Neg\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle B3\rangle,\langle F2\rangle}, 2)$,
 .16      mk- $(\langle Zero\rangle, \langle Zero\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle B2\rangle,\langle F1\rangle}, 2)$,
 .17      mk- $(\langle Zero\rangle, \langle Zero\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2, 2)$,
 .18      mk- $(\langle Zero\rangle, \langle Zero\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle B3\rangle,\langle F4\rangle}, 2)$,
 .19      mk- $(\langle Zero\rangle, \langle Pos\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle B2\rangle,\langle F3\rangle}, 2)$,
 .20      mk- $(\langle Zero\rangle, \langle Pos\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle B1\rangle,\langle F3\rangle}, 2)$,
 .21      mk- $(\langle Zero\rangle, \langle Pos\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle B1\rangle,\langle F4\rangle}, 2)$,
 .22      mk- $(\langle Pos\rangle, \langle Neg\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle F1\rangle}, 2^{\langle F2\rangle,\langle F3\rangle})$,
 .23      mk- $(\langle Pos\rangle, \langle Neg\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle F1\rangle,\langle F2\rangle}, 2)$,
 .24      mk- $(\langle Pos\rangle, \langle Neg\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle F2\rangle}, 2^{\langle F1\rangle,\langle F4\rangle})$,
 .25      mk- $(\langle Pos\rangle, \langle Zero\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle F1\rangle,\langle F3\rangle}, 2)$,
 .26      mk- $(\langle Pos\rangle, \langle Zero\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle F2\rangle,\langle F3\rangle}, 2^{\langle F1\rangle,\langle F4\rangle})$,
 .27      mk- $(\langle Pos\rangle, \langle Zero\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle F2\rangle,\langle F4\rangle}, 2)$,
 .28      mk- $(\langle Pos\rangle, \langle Pos\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle F3\rangle}, 2^{\langle F1\rangle,\langle F4\rangle})$,
 .29      mk- $(\langle Pos\rangle, \langle Pos\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle F3\rangle,\langle F4\rangle}, 2)$,
 .30      mk- $(\langle Pos\rangle, \langle Pos\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle F4\rangle}, 2^{\langle F2\rangle,\langle F3\rangle})$
 .31    end;
```

Selection of left, right, up, and down thrusters resulting from Y, Z, and roll commands.

```
50.0  $LRUDThrusters : AUX‘AxisCommand \times AUX‘AxisCommand \times AUX‘AxisCommand$
 .1                    $\rightarrow ThrusterSet \times ThrusterSet$
 .2   $LRUDThrusters\,(A, B, C) \triangleq$
 .3     cases mk- $(A, B, C)$ :
 .4       mk- $(\langle Neg\rangle, \langle Neg\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2, 2)$,
 .5       mk- $(\langle Neg\rangle, \langle Neg\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2, 2)$,
 .6       mk- $(\langle Neg\rangle, \langle Neg\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2, 2)$,
 .7       mk- $(\langle Neg\rangle, \langle Zero\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle L1R\rangle}, 2^{\langle L1F\rangle, \langle L3F\rangle})$,
 .8       mk- $(\langle Neg\rangle, \langle Zero\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle L1R\rangle, \langle L3R\rangle}, 2^{\langle L1F\rangle, \langle L3F\rangle})$,
 .9       mk- $(\langle Neg\rangle, \langle Zero\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle L3R\rangle}, 2^{\langle L1F\rangle, \langle L3F\rangle})$,
 .10      mk- $(\langle Neg\rangle, \langle Pos\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2, 2)$,
 .11      mk- $(\langle Neg\rangle, \langle Pos\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2, 2)$,
 .12      mk- $(\langle Neg\rangle, \langle Pos\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2, 2)$,
 .13      mk- $(\langle Zero\rangle, \langle Neg\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle U3R\rangle}, 2^{\langle U3F\rangle, \langle U4F\rangle})$,
 .14      mk- $(\langle Zero\rangle, \langle Neg\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle U3R\rangle, \langle U4R\rangle}, 2^{\langle U3F\rangle, \langle U4F\rangle})$,
 .15      mk- $(\langle Zero\rangle, \langle Neg\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle U4R\rangle}, 2^{\langle U3F\rangle, \langle U4F\rangle})$,
 .16      mk- $(\langle Zero\rangle, \langle Zero\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle L1R\rangle, \langle R4R\rangle}, 2)$,
 .17      mk- $(\langle Zero\rangle, \langle Zero\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2, 2)$,
 .18      mk- $(\langle Zero\rangle, \langle Zero\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle R2R\rangle, \langle L3R\rangle}, 2)$,
 .19      mk- $(\langle Zero\rangle, \langle Pos\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle D2R\rangle}, 2^{\langle D1F\rangle, \langle D2F\rangle})$,
 .20      mk- $(\langle Zero\rangle, \langle Pos\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle D1R\rangle, \langle D2R\rangle}, 2^{\langle D1F\rangle, \langle D2F\rangle})$,
 .21      mk- $(\langle Zero\rangle, \langle Pos\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle D1R\rangle}, 2^{\langle D1F\rangle, \langle D2F\rangle})$,
 .22      mk- $(\langle Pos\rangle, \langle Neg\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2, 2)$,
 .23      mk- $(\langle Pos\rangle, \langle Neg\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2, 2)$,
 .24      mk- $(\langle Pos\rangle, \langle Neg\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2, 2)$,
 .25      mk- $(\langle Pos\rangle, \langle Zero\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2^{\langle R4R\rangle}, 2^{\langle R2F\rangle, \langle R4F\rangle})$,
 .26      mk- $(\langle Pos\rangle, \langle Zero\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2^{\langle R2R\rangle, \langle R4R\rangle}, 2^{\langle R2F\rangle, \langle R4F\rangle})$,
 .27      mk- $(\langle Pos\rangle, \langle Zero\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2^{\langle R2R\rangle}, 2^{\langle R2F\rangle, \langle R4F\rangle})$,
 .28      mk- $(\langle Pos\rangle, \langle Pos\rangle, \langle Neg\rangle) \rightarrow$ mk- $(2, 2)$,
 .29      mk- $(\langle Pos\rangle, \langle Pos\rangle, \langle Zero\rangle) \rightarrow$ mk- $(2, 2)$,
 .30      mk- $(\langle Pos\rangle, \langle Pos\rangle, \langle Pos\rangle) \rightarrow$ mk- $(2, 2)$
 .31    end;
```

An integrated six degree-of-freedom command is mapped into a vector of actuator commands. Selection tables provide lists of thrusters and both mandatory and optional thrusters are included as appropriate.

```
51.0  $SelectedThrusters : AUX`SixDofCommand \times AUX`RotCommand \times$
  .1                      $AUX`RotAxis$-set $\times AUX`RotAxis$-set $\rightarrow ThrusterSet$
  .2  $SelectedThrusters\,(hcm, aah, active\text{-}axes, ignore\text{-}hcm) \triangleq$
  .3    let mk-$AUX`SixDofCommand\,(tran, rot) =$
  .4            $IntegratedCommands\,(hcm, aah, active\text{-}axes, ignore\text{-}hcm),$
  .5      mk-$(bf\text{-}mandatory, bf\text{-}optional) =$
  .6            $BFThrusters\,(tran\,(\langle X\rangle), rot\,(\langle Pitch\rangle), rot\,(\langle Yaw\rangle)),$
  .7      mk-$(lrud\text{-}mandatory, lrud\text{-}optional) =$
  .8            $LRUDThrusters\,(tran\,(\langle Y\rangle), tran\,(\langle Z\rangle), rot\,(\langle Roll\rangle)),$
  .9      $bf\text{-}thr =$ if $rot\,(\langle Roll\rangle) = \langle Zero\rangle$
  .10             then $bf\text{-}optional \cup bf\text{-}mandatory$
  .11             else $bf\text{-}mandatory,$
  .12     $lrud\text{-}thr =$ if $rot\,(\langle Pitch\rangle) = \langle Zero\rangle \wedge rot\,(\langle Yaw\rangle) = \langle Zero\rangle$
  .13              then $lrud\text{-}optional \cup lrud\text{-}mandatory$
  .14              else $lrud\text{-}mandatory$ in
  .15   $bf\text{-}thr \cup lrud\text{-}thr$

end $TS$
```

A.5 AAH module - Automatic attitude hold

An automatic attitude hold (AAH) capability may be invoked to maintain near-zero rotation rates. A pushbutton mounted on the hand grip engages AAH with a single button click, and disengages with a double click. Internal state information is maintained to observe the button pushing protocol, and keep track of status for each axis.

```
module $AAH$
      imports
52.0    from $AUX$ all ,
53.0    from $HCM$ all
      exports all
definitions
54.0  state $AAH$ of
  .1    $active\text{-}axes : AUX`RotAxis$-set
  .2    $ignore\text{-}hcm : AUX`RotAxis$-set
  .3    $toggle : EngageState$
  .4    $timeout : \mathbb{N}$
  .5    init $s \triangleq s =$ mk-$AAH\,(2, 2, \langle AAH\_off\rangle, 0)$
  .6  end
types
```

```
55.0  $EngageState = \langle AAH\_off\rangle \mid \langle AAH\_started\rangle \mid \langle AAH\_on\rangle \mid \langle pressed\_once\rangle \mid$
  .1                 $\langle AAH\_closing\rangle \mid \langle pressed\_twice\rangle$
```

values

```
56.0  $click\text{-}timeout : \mathbb{N} = 10$
```

AAH state information is updated in every frame. Key transitions in the engage-state diagram cause various state components to be updated.

operations

```
57.0  $Transition : HCM`ControlButton \times AUX`SixDofCommand \times \mathbb{N} \overset{o}{\to} ()$
  .1  $Transition\,(button\text{-}pos, hcm\text{-}cmd, clock) \triangleq$
  .2    let $engage = ButtonTransition\,(toggle, button\text{-}pos, active\text{-}axes, clock, timeout),$
  .3        $starting = (toggle = \langle AAH\_off\rangle) \wedge (engage = \langle AAH\_started\rangle)$ in
  .4    $(active\text{-}axes := \{a \mid a \in AUX`rot\text{-}axis\text{-}set \cdot$
  .5                            $starting \vee$
  .6                            $(engage \neq \langle AAH\_off\rangle \wedge a \in active\text{-}axes \wedge$
  .7                             $(hcm\text{-}cmd.rot\,(a) = \langle Zero\rangle \vee a \in ignore\text{-}hcm))\};$
  .8     $ignore\text{-}hcm := \{a \mid a \in AUX`rot\text{-}axis\text{-}set \cdot$
  .9                            $(starting \wedge hcm\text{-}cmd.rot\,(a) \neq \langle Zero\rangle) \vee$
  .10                           $(\neg\, starting \wedge a \in ignore\text{-}hcm)\};$
  .11    $timeout :=$ if $toggle = \langle AAH\_on\rangle \wedge engage = \langle pressed\_once\rangle$
  .12                then $clock + click\text{-}timeout$
  .13                else $timeout;$
  .14    $toggle := engage);$
```

```
58.0  $ActiveAxes : () \overset{o}{\to} AUX`RotAxis$-set
  .1  $ActiveAxes\,() \triangleq$
  .2    return $active\text{-}axes;$
```

```
59.0  $IgnoreHcm : () \overset{o}{\to} AUX`RotAxis$-set
  .1  $IgnoreHcm\,() \triangleq$
  .2    return $ignore\text{-}hcm;$
```

```
60.0  $Toggle : () \overset{o}{\to} EngageState$
  .1  $Toggle\,() \triangleq$
  .2    return $toggle$
```

functions

```
61.0  $AllAxesOff : AUX`RotAxis\text{-set} \to \mathbb{B}$
  .1  $AllAxesOff\,(active) \triangleq$
  .2      $active = 2;$
```

On each frame, the sampled value of the AAH engage button is checked to determine whether AAH is engaging or disengaging. This function models the AAH engagement state diagram.

```
62.0  $ButtonTransition : EngageState \times HCM`ControlButton \times AUX`RotAxis\text{-set} \times$
  .1                           $\mathbb{N} \times \mathbb{N} \to EngageState$
  .2  $ButtonTransition\,(estate, button, active, clock, timeout) \triangleq$
  .3      cases mk- $(estate, button)$ :
  .4          mk- $(\langle AAH\_off \rangle, \langle Up \rangle) \to \langle AAH\_off \rangle,$
  .5          mk- $(\langle AAH\_off \rangle, \langle Down \rangle) \to \langle AAH\_started \rangle,$
  .6          mk- $(\langle AAH\_started \rangle, \langle Up \rangle) \to \langle AAH\_on \rangle,$
  .7          mk- $(\langle AAH\_started \rangle, \langle Down \rangle) \to \langle AAH\_started \rangle,$
  .8          mk- $(\langle AAH\_on \rangle, \langle Up \rangle) \to$ if $AllAxesOff\,(active)$
  .9                                            then $\langle AAH\_off \rangle$
  .10                                           else $\langle AAH\_on \rangle,$
  .11         mk- $(\langle AAH\_on \rangle, \langle Down \rangle) \to \langle pressed\_once \rangle,$
  .12         mk- $(\langle pressed\_once \rangle, \langle Up \rangle) \to \langle AAH\_closing \rangle,$
  .13         mk- $(\langle pressed\_once \rangle, \langle Down \rangle) \to \langle pressed\_once \rangle,$
  .14         mk- $(\langle AAH\_closing \rangle, \langle Up \rangle) \to$ if $AllAxesOff\,(active)$
  .15                                                then $\langle AAH\_off \rangle$
  .16                                                elseif $clock > timeout$
  .17                                                then $\langle AAH\_on \rangle$
  .18                                                else $\langle AAH\_closing \rangle,$
  .19         mk- $(\langle AAH\_closing \rangle, \langle Down \rangle) \to \langle pressed\_twice \rangle,$
  .20         mk- $(\langle pressed\_twice \rangle, \langle Up \rangle) \to \langle AAH\_off \rangle,$
  .21         mk- $(\langle pressed\_twice \rangle, \langle Down \rangle) \to \langle pressed\_twice \rangle$
  .22     end
```

end AAH