**A Case Study in Integrating Formal Verification Tools
using the OMDoc
Standard for Mathematical Documents**

by
Xiao Bing Jin

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

September, 2004

© Copyright
2004, Xiao Bing Jin

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-97455-3

Our file Notre référence

ISBN: 0-612-97455-3

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

Abstract

Different formal methods have been advocated in academic research and now even applied in many industrial projects from different aspects. As computer systems become more complex, integration among different formal methods is necessary and can be helpful in providing a correct and rigorous system. All formal methods are based on mathematical symbols and knowledge. OMDoc (Open Mathematical Documents), as an XML application, presents a content markup scheme for mathematical notations and concepts. OMDoc provides a possible infrastructure to serve as a content language for communication of different verification systems based on formal methods.

As a case study, in this thesis, we present an integration solution of two well-known formal verification systems -- SPARK and PVS by translating the SPARK verification language FDL into PVS via the OMDoc approach automatically. The integration solution includes a translator that can translate the SPARK FDL syntax into OMDoc formats, and an updated interface that can transform the OMDoc format into PVS syntax. Moreover, as intermediate documents, the generated OMDoc format can be transformed into other languages. The main contribution of the thesis is as a case study in the use of the OMDoc standard. We demonstrate some limitations of the OMDoc approach.

Acknowledgements

I would like to sincerely thank my supervisor Dr. Doug Howe for this guidance and support of my work. I must also appreciate to Stephen Michell of National Research Council for his help in providing valuable SPARK code. I would especially like to thank my friend, Paul Boone for his comments and proofreading.

Finally many thanks to my parents, my wife Xiaoli, and our family members, for all their love and support to complete of this thesis.

Contents

Chapter 1	Introduction	1
1.1	Background.....	1
1.2	Problems and Motivation	3
1.3	The Contributions of the thesis	6
1.4	Organization of the Thesis	8
Chapter 2	SPARK FDL	10
2.1	The SPARK language and tools	10
2.2	The SPARK FDL Language	14
2.2.1	The Examiner output files.....	14
2.2.2	FDL as a formal verification language.....	16
2.2.3	FDL reserved words, Numeric Literals and Identifiers	16
2.2.4	FDL Declarations	17
2.2.4.1	Types Declarations	17
2.2.4.2	Constant Declarations	20
2.2.4.3	Variable Declarations	20
2.2.4.4	Function Declarations	20
2.2.5	FDL Expressions.....	21
2.2.5.1	Operators	21
2.2.5.2	Function-designators	22
2.2.5.3	Quantifiers	24
2.2.5.4	Sequences.....	24
2.2.5.5	Sets.....	25
2.2.5.6	Attributes	25
2.2.6	The FDL rules file	26
2.2.7	The FDL Verification Conditions file	29
Chapter 3	The PVS Language	33

3.1 PVS Overview	33
3.2 The PVS Language	34
3.2.1 Identifiers, reserved words and comments	34
3.2.2 Declarations	34
3.2.2.1 Type Declarations	34
3.2.2.2 Constant Declarations	36
3.2.2.3 Variable Declaration	37
3.2.2.4 Formula Declarations	37
3.2.3 PVS Expressions	37
3.2.3.1 Boolean expressions	38
3.2.3.2 Numeric expressions	38
3.2.3.3 Function expressions	38
3.2.3.4 Binding expressions	38
3.2.3.5 Record Expressions	39
3.2.3.6 Record Accessors	39
3.2.3.7 Override Expressions	39
3.3 The PVS specification file	40
Chapter 4 OpenMath	41
4.1 Overview	41
4.2 The OpenMath Objects and Encodings	42
4.2.1 Basic OpenMath Objects and Encodings	42
4.2.2 Compound OpenMath Objects and Encodings	43
4.3 Content Dictionaries	45
Chapter 5 OMDoc	48
5.1 Overview	48
5.2 OMDoc Statements	50
5.2.1 Symbol Declaration	51
5.2.2 Type Declaration	51
5.2.3 Definition	52
5.2.4 Axiom	53
5.2.5 Assertion	54
5.3 Theory	54

5.4 OMDoc Document and locating	56
5.5 Transforming OMDoc.....	58
Chapter 6 The Translation	59
6.1 Overview	59
6.2 Translating FDL into the generic OMDoc.....	64
6.2.1 Translating FDL constant declarations	65
6.2.2 Dealing with FDL variable declarations.....	68
6.2.3 Translating FDL type declarations.....	71
6.2.4 Dealing with FDL rule declarations.....	74
6.2.5 Translating FDL Verification Conditions.....	78
6.3 Translating the Generic OMDoc to the PVS-oriented OMDoc.....	81
6.3.1 Changing attribute values of <OMS> elements	82
6.3.2 FDL mod operator.....	82
6.4 Transforming from PVS-Oriented OMDoc into PVS	83
6.5 Implementation	85
6.5.1 Tools, Language and XML Processing Approach.....	85
6.5.2 Implementation details.....	87
Chapter 7 Conclusions.....	93
7.1 Summaries and limitations.....	93
7.2 Future work	96
Bibliography	97
Appendices.....	101
A. FDL reserved Words.....	101
B. PVS reserved Words.....	101
C. An example of the generic OMDoc.....	102
D. An example of the PVS-oriented OMDoc	107

List of Tables

Table 2.1: SPARK Examiner's output files.	14
Table 2.2: FDL Boolean operators	21
Table 2.3: FDL Sets operations.....	25
Table 2.4: FDL Expression for function Attributes	26
Table 5.1: Symbol, definition, axiom, assertion and theory in OMDoc.	50

List of Figures

Figure 2.1 A simple example of SPARK program SwapElement.ad	12
Figure 2.2 The proof declarations for SwapElement	13
Figure 2.3 The proof rules for SwapElement	13
Figure 2.4 The verification condition for SwapElement	13
Figure 2.5 The FDL overall proof process and the relationship of SPARK tools	15
Figure 2.6 An example of a FDL record type declaration.	19
Figure 2.7 An example of a FDL rules .rls file fragment	28
Figure 2.8 A simple example of SPARK program IntRoot.ad	31
Figure 2.9 A simple example of SPARK FDL .vcg file with 3 VCs.	32
Figure 3.1 An example of PVS specification file.	40
Figure 5.1 An example of a symbol and a type declaration in OMDoc	51
Figure 5.2 An example of a symbol and its definition in OMDoc.	52
Figure 5.3 An example of a primitive symbol and its axiom definition in OMDoc.	53
Figure 5.4 A TRUE assertion declaration in OMDoc	54
Figure 5.5 An OMDoc fragment theory definition	55
Figure 6.1 A high-level diagram of translation from SPARK FDL into PVS	63
Figure 6.2 An FDL Declarations file (swapelement.fdl).	65
Figure 6.3 An FDL rules file (swapelement.rls)	65
Figure 6.4 An FDL Verification Conditions file (swapelement.vcg)	66
Figure 6.5 The translation result from an FDL constant into an OMDoc form	67
Figure 6.6 The OMDoc fragment for FDL variable declarations and their binding information.	71
Figure 6.7 The OMDoc fragment for the FDL record type declaration.	73
Figure 6.8 The OMDoc fragment for the FDL array type declaration.	74
Figure 6.9 An example of the SPARK FDL rules declaration.	75
Figure 6.10 The PVS language representation for swapelement theory.	84

Figure 6.11 Parsing FDL Declarations (.fdl file).	88
Figure 6.12 Parsing FDL rule file (.rls file).....	89
Figure 6.13 Parsing FDL Verification Conditions file (.vcg file).	90
Figure 6.14 Parsing FDL Language and generating OMDoc files.....	92

Chapter 1 Introduction

We live in a chaotic world. While computer systems dominate our daily lives, they wake us up in the morning, help cook our food, control traffic systems, provide communication systems in our social lives, entertain us and even give us cash (if needed). The evolutions of computer technologies are supposed to make things much easier, but in some way they do not. Sometimes computer systems work improperly, make mistakes or even cause failures. For application-critical systems, such anomalies could cause us great inconveniences or financial loss. For mission-critical or safety-critical systems, such failures could even conceivably result in catastrophes, causing loss of life.

As computer systems are pervasive in our society, the size and complexity of systems increase dramatically. The need to be assured of the correctness of the system has also become increasingly vital. This is especially the case for the designing of safety-critical systems, because most safety-critical systems have their own standards to meet. Some of the software-related standards are: the Department of Defence (DoD)'s DOD-STD-2167A [1] in the USA, and the Atomic Energy Control Board (AECB)'s 2.234.1 [2] in Canada. However, the traditional coding and testing software lifecycle approach alone does not satisfy these higher standards and the demands of safety-critical systems, because these systems must be specified and verified with as much rigour and formality as possible.

1.1 Background

Formal methods have been widely advocated and accepted for safety-critical systems, and their usage is being suggested in an increasing number of standards in the safety-critical

domain [2]. The term *formal methods*, is primarily used for specifying and verifying the properties of a system based upon mathematical symbols and logic. A formal method usually contains *a formal language* and *a formal reasoning system* [46]. A formal language is a specification language with precisely defined syntax and semantics [46]. A formal reasoning system includes a formal language and associated proof tool(s).

Over the years, there have been a number of specification languages, methods and tools based on formal methods available in both academic and industrial fields. They have been applied to various aspects of design and development of systems. For instance, Specification and Description Language (SDL) [3], Vienna Development Method (VDM) [4], and Z notation [5] are often used primarily for requirements specification; Symbolic Model Verifier (SMV) [6] and SPIN [7] are automated verification tools for finite state systems; Prototype Verification System (PVS) [8] is another reasoning verification system; and SPARK [9], is a subset of Ada [15] language with a several static analysis tools based on the Z notation. Moreover, there have been a number of successful stories [29] in both safety-critical and commercial application systems that applied formal methods for specifying and verifying the correctness of systems. Some of the notable examples include: Lockheed Corporation has applied SPARK to analyze the software code for Lockheed C130J aircraft and IBM has used Z notation to formalize its Customer Information Control System (CICS), an online transaction processing system with thousands of installation worldwide [29].

So far, formal methods have been most promisingly successful in the system specification and verification domains. In general, a formal specification can be achieved through using a formal language to describe the properties of the specified system. SPARK, as mentioned above, is probably one of the most successfully used formal languages to specify the functional behaviours of a system. SPARK includes a formal specification language, namely *SPARK language*. The SPARK language is a subset of the Ada programming language with additional annotations. The use of SPARK in real-world projects (e.g. as illustrated above) as well as other experience in industries [47] has proven the success of formal specifications.

In system verification area, formal methods go further beyond the specification by using verification methods to analyze whether the description of a system satisfies its specification.

There are two well-known verification approaches in formal methods fields: *model checking* and *theorem proving*. Model checking is an automatic verification method to verify desired properties of a system. Unlike traditional simulation and testing techniques, model checking builds on finite model systems and therefore the verification is exhaustive. SMV and SPIN are two model checkers based on model checking technique. In contrast to model checking, theorem proving, as another verification technique, can directly deal with the infinite state problem. It usually uses mathematical formulae, axioms and inference rules to deduce the desired properties. SPARK, aside from its specification language, also includes a formal reasoning system based on the theorem proving technique. The SPARK reasoning system includes a set of tools containing a theorem generator and semi-automatic verification tools. The theorem generator can generate theorems based on the SPARK specifications. In addition, the verification tools in SPARK can be used to prove the soundness of these theorems in order to verify with the corresponding specifications. Unfortunately, SPARK can only deal with static, functional behaviours. The theorem proving technique that is applied by SPARK is relatively primitive, it cannot be used to verify the properties of run-time, non-behaviours (such as performance and real-time constraints). Another widely used verification system is the PVS system. Similar to SPARK, PVS also includes a specification language and a semi-automatic verification tool. Unlike SPARK that only deals with static analysis and functional behaviours, PVS can be applied to verify both sequential code and concurrent behaviour of a system.

1.2 Problems and Motivation

As briefly discussed above, each of the commonly used formal methods has its own advantages and disadvantages. No one formal method is the best for any given situation. Different techniques must be applied at different stages based on different purposes. For particular classes of systems, some are more appropriate than others. Therefore, a practical approach is to use different methods or tools in combination. This is especially the case while specifying and verifying a large complex system.

In recent years, the integration of formal methods has become a popular topic [29, 46, 49]. The integration from semantic and syntactic level among different formal specification languages is one of the fundamental and promising directions in the methods integration domain. A number of proposals have been presented and some of them have been implemented [48, 50, 51, 52, 53, 54]. However, all of these efforts are point-to-point translations (i.e. features from one specification language to another). The integration results are functional, but they are tightly coupled to the particular specification languages and are arised the problems such as interoperability, distribution and modularization. The translation lacks of an open standard, or a universal interface.

All formal methods techniques, including specification languages and verification methods, are based on mathematical knowledge. Mathematics is the basis of all these techniques. In the past, the way of doing mathematics was dominated by hand. But now, web-based mathematical systems provide an infrastructure that supports all aspects of doing mathematics. Several mathematical interchange formats have been proposed and some of them have become standards. One of them, the Mathematics Markup Language (MathML) [11, 35] is the first Worldwide Web Consortium (W3C) [12] recommendation for the Extensible Markup Language (XML) [13] for presenting mathematics over the web. On the other hand, instead of focusing on the presentation level of mathematics, OpenMath [22, 34], another XML application, proposes a series of standards for the rich contents of mathematical objects from the semantic level. OpenMath uses specific XML notations that are called *OpenMath objects* and an extension mechanism, namely the *Content Dictionary (CD in short)* to encode mathematical symbols and expressions. In addition, the Open Markup Format for Mathematical Documents (OMDoc) [14], as an extension of OpenMath, further extends the mathematical objects and contents from document and theory level. This is needed because OpenMath does not provide sufficient markup schemes for mathematical concepts like definitions, theorems and theories. Similar to OpenMath, an *OMDoc document* can be considered as a XML document with a special markup scheme to express rich mathematical symbols and expressions especially from the mathematical theory and proof level.

As XML applications, one of the original goals of the OpenMath, MathML and OMDoc is to provide a communication language for mathematical software systems [14]. To accomplish this goal, MathML from the presentation level provides a standard markup scheme for mathematical formulae. OpenMath aims from the semantic level to provide a general markup language for mathematical objects. Moreover, by adding the markup scheme for mathematical statements and theories, OMDoc attempts to provide a universal accepted communication language standard (or *an interlingua*) to support for automated reasoning systems and formal specification systems, so that the interoperability for n systems can be achieved ideally with only $2n$ translations instead of n^2 [14]. To achieve this, OMDoc has even provided several interfaces in the formats of the Extensible Stylesheet Language (XSL) [19] style sheet language. These interfaces can be used to transform an OMDoc document into other formats [14, 28, and 36]. Those other formats include HTML, DHTML and LaTeX for presentation purpose in human readable form, as well as some specification languages for software systems, especially theorem proving systems, such as INKA, TPS and PVS etc [14]. As new emerging methodologies, OpenMath and OMDoc are still developing. To our knowledge, there has been little work on evaluating how well these formats actually work for the “real-world” specification and verification programming languages. One of the goals of this thesis is to study the effectiveness of these leading interchange formats (OpenMath and its extension OMDoc) for formal specification and verification languages.

As introduced earlier, in formal specifications area, the SPARK system has been widely used in industries to design, develop and verify complex safety-critical systems to ensure the requirements meet the industrial regulations. The SPARK language provides a precise and unambiguous meaning to specify the functional behaviors of systems. SPARK associated tools not only can act as a typical compiler to conduct the static semantic analysis and run-time error checking, more importantly, they can also perform the program verification analysis. The verification analysis can be done in two steps. The first step, the SPARK tools can generate various declarations, rules and corresponding theorems from the SPARK code. These generated declarations, rules and theorems are expressed in a form, which is called *the Functional Description language* (FDL) modeling language in SPARK [21]. The second step, the verification tools in SPARK can go further to verify by proof whether the theorems are conformed to the specified properties declared in the SPARK code or not. Unfortunately, as

indicated earlier, the verification technique provided by SPARK is relatively simple and primitive, thus it only supports to analyze and verify properties for the sequential SPARK program. Moreover, the verification language SPARK FDL is based on *first-order logic* [24] rather than richer *higher-order logic* [10,25]. In other words, FDL supports only simple propositional logics and predicate logics, such as relations, quantifiers; it does not support the features of higher-order logic, such as functions can take functions as arguments, and quantifiers can be applied as variables of functions etc. Therefore, even for functional verifications, SPARK has its limitations as well.

Most recently, SPARK has added an extension, namely *the SPARK Ravenscar Profile* [16] to deal with some deterministic concurrent real-time issues, but such extension still does not satisfy the program with the non-deterministic behaviours such as timing deadlines [17] as well as many other concurrent issues (e.g. protected objects, pre-emption and resource usage) addressed in [18]. Other verification methodologies, such as theorem proving or model checking techniques are needed for further reasoning about those non-behavioural properties. The PVS system, as briefly stated before, as one of the most widely used verification systems can be integrated with the SPARK system to conduct the further verification for the non-behavioural properties. Besides, PVS is based on higher-order logic; it contains much richer logic than first-order logic. Even for verifying functional behaviours, PVS is more powerful than SPARK. Therefore, another goal of this thesis is to achieve the integration between the SPARK and PVS systems from the semantic level by applying the OMDoc approach.

1.3 The Contributions of the thesis

The two goals of this thesis are achieved through a case study of using mathematical interchange formats -- OpenMath and OMDoc to translate the SPARK FDL verification language into the PVS language. The contributions can be considered as twofold:

Primarily, a translation solution from the SPARK FDL language into the PVS language via the OMDoc approach is presented. Based on the OpenMath and OMDoc methodologies, we have implemented a translator that can automatically parse the SPARK FDL syntax and generate the corresponding OMDoc document as outputs. In addition, the generated OMDoc documents can be automatically transformed into the valid PVS syntax. More importantly, the generated OMDoc documents as an intermediate format can be considered as a universal communication language to be transformed into other formal specification verification languages.

Our translation solution demonstrates that, the integration of two verification systems from SPARK to PVS can be achieved by applying the OMDoc methodology. The translation demonstrates that: as the intermediate results of the translation, OMDoc documents can provide an open standard format to represent the mathematical concepts and knowledge as it has claimed. In general, from our case study presented in Chapter 6, we can draw that from the semantic and syntactical levels, theoretically; the translation among different formal languages can be achieved via the OMDoc approach as well. As a result, ideally, the interoperability among different theorem proving systems can be accomplished.

On the other hand, through our case study we also have discovered a number of shortcomings of the OMDoc approach. First, although FDL and PVS are specification and verification languages, they also contain several data types similar to other programming languages, such as enumerations, arrays and records etc. OpenMath and its extension OMDoc, however, are mainly dealt with mathematical objects, symbols and expressions rather than specification and verification languages. So originally, OpenMath and OMDoc do not have an explicit way to express such data types. These kinds of data types, however, are fundamental in the SPARK FDL and the PVS as well as other specification and verification languages. Therefore, we have made some customizations based on the OpenMath and OMDoc mechanism to support these data types.

As introduced earlier, OMDoc claimed that the OMDoc document provided a universal accepted language standard or an interlingua for formal specification and reasoning systems, so that the interoperability for n systems can be achieved ideally with only $2n$ translations

instead of π^2 . Moreover, OMDoc included an XSLT style sheet interface for PVS and it claimed that the interface could transform OMDoc documents into the PVS syntax. But surprisingly, we have found that the interface used different notations other than those of the standard OpenMath and OMDoc, so a standard OMDoc document has to be transformed as a variant OMDoc form, otherwise it cannot even be recognized by the interface. This inconsistent deficiency dramatically weakened OMDoc as an interlingua.

Another flaw of OMDoc is that the interface that is used to transform an OMDoc document to the PVS syntax provided by OMDoc also contains inconsistent markup scheme for mathematical symbols and definitions. Also, the interface does not support those data types that are described in earlier.

As a result, the second main contribution of this thesis can be summarized as follows:

- Define several customized markup schemes in the translator based on the OpenMath and OMDoc mechanism
- Generate OMDoc documents as an interlingua from SPARK FDL
- Provide an updated version of the interface in order to transform properly them into the correct of the PVS syntax.

1.4 Organization of the Thesis

The organization of this thesis is as follows:

Chapter 2 starts with a brief description of SPARK, its associated tools and the rationale of SPARK. This chapter also briefly describes how SPARK FDL is generated by SPARK tools. It then, mainly from a semantic and syntactic level, systematically describes the SPARK FDL verification language in detail. Similar to Chapter 2, Chapter 3 first gives an overview of the PVS system and mainly discusses the declarations and expressions of the PVS language.

Chapter 4 introduces the OpenMath approach, depicts OpenMath objects, encoding and the Content Dictionary mechanism. Chapter 5 expands the contents of Chapter 4 by introducing the OMDoc approach. It addresses how OMDoc extends the OpenMath methodology to support mathematical concepts, such as symbol, definition, axiom, theorem and theory etc. by introducing some new OMDoc elements. It also briefly describes with summaries on how an OMDoc document can be transformed into other formats.

Chapter 6 contains most of contributions of this thesis. It starts with the overview of the translation via the OMDoc approach. It discusses the benefits of the OMDoc approach and briefly introduces the steps of the translation. It then mainly addresses the difficulties of the translation, the problems that are encountered during the experimental case study and our solutions respectively. The last section of Chapter 6 illustrates a sample result of the translation and the detailed implementation.

The final Chapter 7 concludes with summaries and limitations as well as future research.

Chapter 2 SPARK FDL

2.1 The SPARK language and tools

SPARK is a high-level programming language from Praxis Critical Systems Limited [20], UK, designed for writing software for safety-critical systems. SPARK comprises a kernel that is subset of Ada language, with additional annotations that make it possible to carry out program analysis, run-time error checking, and verifying program correctness by using the associated tools with the language: the SPARK Examiner, Simplifier and Proof Checker [21].

The rationale of SPARK is: as a generic programming language, *the Ada programming language* is not suitable for designing rigorous systems due to its ambiguous definitions and excessive complexities. For the development of safety-critical systems, there are needs for a “safe subset” of Ada to assure the correctness of systems. SPARK exploits the strengths of Ada while eliminating all its potential ambiguities and insecurities. SPARK’s additional annotations, in the form of Ada comments that are ignored by a standard Ada compiler, contain extra information regarding specifications and requirements for analysis. These annotations, together with the SPARK core Ada subset, can be processed by the SPARK tool -- Examiner. The Examiner, first acts as a compiler to perform the static analysis of each compilation unit. The static analysis may include lexical and syntactic analysis, semantic checking, and data and information flow analysis. More importantly, the Examiner can generate theorems based on SPARK code and annotations. These theorems can be further

discharged or proved by using other SPARK tools – Simplifier and Proof Checker. Once proven, the SPARK program is demonstrated that the specified properties are hold and the code conforms to its specification.

A theorem is referred to as a *verification condition* (VC in short) in SPARK. Aside from theorems or VCs, the Examiner also generates the corresponding proof declarations and assistant rules in order to prove these theorems. All of these, proof declarations, rules and VCs are called *proof obligations* in SPARK. The proof obligations are expressed in a form, called the FDL(*Functional Description Language*) modeling language in SPARK [21].

SPARK annotations have the following forms:

- Global definitions – declare imported or exported global variables in procedures or functions subprogram.
- Dependency relations – specify the information flow between imported variables and exported variables
- Proof contexts – includes preconditions and postconditions of subprograms¹, and assertion statements.

The first two annotations are mandatory specifications for static and semantic analysis. Although it is non-mandatory, proof contexts can be used by the Examiner to generate theorems. In here, preconditions and postconditions are predicates to constrain variables of subprograms.

In SPARK, anything followed by the three characters “--#” is considered to be the start of an annotation. The following example illustrates a SPARK program with annotations:

¹ Similar to Ada, a subprogram in SPARK is a program unit whose execution is invoked by a subprogram call. Subprograms have two forms: procedures and functions.

```

Line
1  package P
2  is
3      type Index is range 1..10;
4      type Atype is array(Index) of Integer;
5  end P;
6
7  package body P
8  is
9      procedure SwapElement(I, J : in Index;
10         A : in out Atype)
11         --# derives A from A, I, J;
12         --# post A = A~[I => A~(J); J => A~(I)];
13         is
14             temp : Integer;
15         begin
16             temp := A(I);
17             A(I) := A(J);
18             A(J) := temp;
19         end SwapElement;
20 end P;

```

Figure 2.1 A simple example of SPARK program SwapElement.ada

The SwapElement.ada routine in Figure 2.1 takes an array and exchanges the two elements selected by indices passed in as parameters. Two kinds of SPARK annotations: dependency relations and postcondition are illustrated in line 11 and 12 respectively. The postcondition in line 12 can be read as: *array A is the original array A with its I^{th} element replaced by its original J^{th} element and its J^{th} element replaced by its original I^{th} element* [30]. Also, in SPARK a tilde ($x\sim$) is used to decorate the initial value of an imported, exported variable (apart from in the precondition, where all values are initial) [30]. The full description of the SPARK language is outside of this thesis' scope. Please refer to “SPARK95 user manual” [23] and John Barnes's book, “High Integrity Ada, The SPARK Approach” [21] for a complete study.

Running the SPARK Examiner with SwapElement.ada produces the proof obligations for the *procedure SwapElement*. The proof obligations are illustrated in the following three figures respectively:

```

title procedure swapelement;

function round__(real) : integer;

```

```

type atype = array [integer] of integer;
type index = integer;
const index__last : integer = pending;
const index__first : integer = pending;
const integer__last : integer = pending;
const integer__first : integer = pending;
var a : atype;
var j : integer;
var i : integer;

end;

```

Figure 2.2 The proof declarations for SwapElement

```

rule_family swapelement_rules:
  X requires [X:any] &
  X <= Y requires [X:ire, Y:ire] &
  X >= Y requires [X:ire, Y:ire].

swapelement_rules(1): character__pos(X) may_be_replaced_by X.
swapelement_rules(2): character__val(X) may_be_replaced_by X.
swapelement_rules(3): integer__first <= integer__last may_be_deduced.
swapelement_rules(4): integer__base__first <= integer__base__last may_be_deduced.
swapelement_rules(5): integer__base__first <= integer__first may_be_deduced.
swapelement_rules(6): integer__base__last >= integer__last may_be_deduced.
swapelement_rules(7): index__first may_be_replaced_by 1.
swapelement_rules(8): index__last may_be_replaced_by 10.
swapelement_rules(9): index__base__first <= index__base__last may_be_deduced.
swapelement_rules(10): index__base__first <= index__first may_be_deduced.
swapelement_rules(11): index__base__last >= index__last may_be_deduced.

```

Figure 2.3 The proof rules for SwapElement

```

For path(s) from start to finish:
procedure_swapelement_1.
H1: true.
H2: i >= index__first.
H3: i <= index__last.
H4: j >= index__first.
H5: j <= index__last.
H6: for_all (i__1: index, ((i__1 >= index__first) and (i__1 <= index__last)) -> ((element(a,
[i__1]) >= integer__first) and (element(a, [i__1]) <= integer__last))) .
->
C1: update(update(a, [i], element(a, [j])), [j], element(
a, [i])) = update(update(a, [i], element(a, [j])), [
j], element(a, [i])) .

```

Figure 2.4 The verification condition for SwapElement

Figure 2.4 shows the generated theorem (only one theorem in this case) by the Examiner

program. Because the SPARK source file `SwapElement.adb` contains an array data type, the generated theorem or VC looks awful. Figures 2.2 and 2.3 show the corresponding generated proof declarations and rules that are more readable for human beings. The theorem, together with declarations and rules can be proven by human reasoning, which usually can be tedious and unreliable, or by using theorem proving tools, such as the SPARK Simplifier and the Proof Checker. The examples illustrated from Figures 2.1 – 2.4 give us a general idea how the SPARK Examiner generates the proof obligations, and how the proof obligations are mapped from SPARK Ada and annotations. The next sections will explain and discuss this in detail.

2.2 The SPARK FDL Language

2.2.1 The Examiner output files

As mentioned earlier, the SPARK Examiner generates the proof obligations in the form of which is called the FDL modeling language. For each SPARK subprogram, the proof obligations in the form of the FDL language generated by the SPARK Examiner are actually saved in the following three output files separately:

File	Extension
FDL Declarations	.fdl
Rules	.rls
Analysis (theorems or VCs)	.vcg

Table 2.1: SPARK Examiner’s output files.

The Examiner has naming mechanism to keep the names of FDL output files and the names of subprograms from which they are generated consistent. For instance, for the subprogram *SwapElement* illustrated in Figure 2.1, the corresponding output files are *SwapElement.fdl*, *SwapElement.rls* and *SwapElement.vcg* illustrated in Figures 2.2, 2.3 and 2.4 respectively.

The following figure illustrates the overall proof process and the relationship of SPARK tools.

Figure 2.5 The FDL overall proof process and the relationship of SPARK tools.

2.2.2 FDL as a formal verification language

Unlike SPARK, FDL is a specification and modeling language, not a programming language. FDL uses *first-order logic* [24] to model and describe the behaviours of properties. In other words, as a first-order logic, not only can FDL have expressions to express booleans, such as ‘true’, ‘false’, ‘and’, ‘or’ and ‘xor’ etc, but also can deal with two additional important logical operators: the universal and existential quantifiers. In FDL, they are denoted by the ‘for_all’ and ‘for_some’ reserved words respectively. Actually, we have seen ‘for_all’ in Figure 2.4.

Another important issue needed to be addressed is, as a specification and modeling language, FDL concerns the declarations (including data type, function declarations), rules and verification condition expressions from the mathematical perspective. All of these will be covered in detail in the following sections. Also, the definitions of various FDL declarations are referred in [30]. As for the full description of SPARK FDL language, please refer to [21], [30] and [31].

2.2.3 FDL reserved words, Numeric Literals and Identifiers

As a specification and modeling language, FDL also contains reserved words that are listed in Appendix A.

In FDL, all numeric literals from SPARK are converted to base 10 format.

Example:

16#FFFF_FFFF#	becomes 4294967295
1e3	becomes 1000

All FDL identifiers mapped from SPARK become the lower case and replacing the period in any package prefix by a double underscore to avoid a conflict with any SPARK identifier.

Examples:

Speed becomes speed

Time.Started becomes time__started

2.2.4 FDL Declarations

The proof declarations generated by the SPARK Examiner are stored in the *.fdl* file. FDL declarations are comprised of the following declarations: types, constants, variables, and functions.

2.2.4.1 Types Declarations

FDL has two kinds of types: *simple types* and *structured types*. A simple type is a type whose values have no components, and form an ordered set. The simple types comprise the boolean, integer, real and enumeration types. A structured type is a composite type, characterized by the type(s) of its components and the structuring method. The structured types include the array and record types.

An FDL type-declaration associates an identifier with a type; the identifier may then be used to denote this type. A type-declaration may also give a definition of the type, which refers to the type is said to be concrete; alternatively, the type-definition may be pending, denoted by the reserved word ‘pending’, which means the type is said to be abstract. Both simple types and structured types can be either concrete or abstract.

The integer-type

The type identifier *integer* denotes the integer type. All kinds of SPARK integers, such as negative and positive integers, are mapped as FDL integers.

Example: type x = integer;

The real-type

The type identifier *real* denotes the real type. Similar to the integer type, all SPARK real, fixed and floating point types are modelled as FDL real numbers, in the mathematical sense.

Example: type *y* = real;

The boolean-type

The type identifier *boolean* denotes the Boolean type, whose values are ‘true’ or ‘false’.

Example: type *t* = boolean;

The enumeration-type

Similar to SPARK, an enumeration-type-definition defines an ordered set of values by enumeration of the identifiers that denote these values. The ordering of the values is determined by the sequence in which their identifiers are enumerated, i.e. if *x* precedes *y* then *x* is less than *y*. Unlike integer type, FDL enumeration type simply replaces its equivalent SPARK enumeration type.

Example:

type colour = (red, orange, yellow, green, blue, indigo, violet);

The array-type

An array type is a structured type consisting of a fixed number of elements. An element of an array is designated by one or more index values belonging to declared enumerated or subrange types.

The following example taken from Figure 2.1 and 2.2 illustrates how a SPARK array type

maps to a FDL array type: line 3 and 4 in Figure 2.1 define an array type in SPARK:

```
3 type Index is range 1..10;  
4 type Atype is array(Index) of Integer;
```

In Figure 2.2, a FDL array type mapped from SPARK

type atype = array [integer] of integer;

The record-type

A record type is also a structure type consisting of a fixed number of fields. The record type definition specifies for each field, an identifier and a type.

Examples:

Given a record type in SPARK

```
type daynumber = 1 .. 31;  
type monthname = (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);  
type yearnumber = 0 .. 2000;
```

```
type date is  
record  
  day: daynumber;  
  month: monthname;  
  year: yearnumber;  
end record;
```

the corresponding SPARK FDL record type will be:

```
type date = record  
  day : daynumber;  
  month : monthname;  
  year : yearnumber  
end;
```

Figure 2.6 An example of a FDL record type declaration.

It should be pointed out that FDL uses semicolons ‘;’ as separators, but if a field of record has no semicolon followed by, it implies that field is the last field of the record type definition.

2.2.4.2 Constant Declarations

The constant identifier *const* denotes a constant. The FDL constants produced by the Examiner are always declared as ‘pending’, as we have seen in Figure 2.2.

Example: `const pi real = pending;`

The actual values for such constants are substituted by using suitable, automatically-generated proof rules, which are stored in FDL rules files, as we have seen in Figure 2.3. The FDL rules files will be discussed in the later section.

2.2.4.3 Variable Declarations

The variable identifier *var* denotes a variable. A variable-declaration consists of a list of identifiers denoting new variables, followed by their type.

Example: `var x = real;`

2.2.4.4 Function Declarations

The function identifier *function* denotes a function. FDL functions are pure mathematical functions, as noted earlier. Therefore, unlike SPARK, there are no procedures declarations in FDL.

Example:

```
type stack = pending;
const emptystack : stack = pending;
function push(stack, integer) : stack;
function pop(stack) : stack;
function top(stack) : integer;
function is_empty(stack) : boolean;
function is_full(stack) : boolean;
```

As a summary, a complete FDL declarations file is illustrated in Figure 2.2.

2.2.5 FDL Expressions

FDL expressions consist of operands, operators, function-designators, quantifiers, sets, sequence-constructors and attributes.

2.2.5.1 Operators

The standard FDL operators are: *arithmetic operators*, *boolean operators*, *relational operators*.

Arithmetic operators:

multiplying-operator ::= * | / | div | mod | ** (exponentiation)

adding-operator ::= + | -

relational-operator ::= = | <> | < | > | <= | >=

unary-minus-operator ::= -

Boolean operators:

operator	operation
not	logical negation
and	logical conjunction
or	logical disjunction
->	logical implication
<->	logical equivalence

Table 2.2: FDL Boolean operators

The *not* operator is a unary operator and the others are binary operators.

Relational operators

relational-operator ::= = | <> | < | > | <= | >=

2.2.5.2 Function-designators

A function-designator specifies the evaluation of a function. It consists of the identifier designating the function, and a list of actual parameters. In FDL, besides user declared functions specified in FDL declarations files, there are a number of predefined functions. For convenience, they are designated in different categories.

Arithmetic functions

$\text{abs}(x)$ denotes the absolute value of x .

$\text{sqr}(x)$ denotes $x * x$.

Boolean functions

$\text{odd}(x)$. For an expression x of an integer type, $\text{odd}(x)$ is equivalent to the expression $\text{abs}(x)$

$\text{mod } 2 = 1$.

Functions for manipulating arrays

As illustrated in Figure 2.4, FDL supports two built-in functions to access an array: *element* and *update*.

element: the element of an array a with index j may be read by using the function *element*.

Example: $\text{element}(a, [j])$.

update: an element may be updated by using the function *update*.

Example: $\text{update}(a, [j], x)$.

Functions for manipulating records

SPARK FDL uses a pair of functions starting with *fld_* and *upf_* as prefixes to access the fields of a record type.

Examples:

Given a record type *date* declared in Figure 2.6, accessing the fields of the *date* record is illustrated by the following examples:

```
mon := fld_month(dat);
dat := upf_day(dat, 14);
dat := upf_month(dat, jul);
dat := upf_year(dat, 1989);
```

Functions for assembling structured values

For every array or record type *t*, a function *mk__t* can be used to assemble objects from a collection of values. The parameters of *mk__t* are a series of assignments to the elements of an object of type *t* with an optional default value appearing first.

Examples:

```
mk__date(day := 1, month:= march, year := 2004 );
mk__table(0, [1] := 1, [2] := 99);
mk__matrix(0, [1,1] := 1, [2,2] := 2, [3,3] := 1);
```

Therefore, the prefixes, *fld_*, *upf_* and *mk__* are regarded as predefined FDL identifiers, just like those FDL reserved words.

Functions for bitwise

The FDL predefined bitwise logical functions include:

```
bit__and(I, J);
bit__or(I, J);
bit__xor(I, J);
```

Functions for enumeration

$\text{pred}(x)$: denotes the value immediately preceding that of x in an enumeration of type.

If x is the least member of T then the value of $\text{pred}(x)$ is undefined.

$\text{succ}(x)$: denotes the value immediately succeeding that of x in an enumeration of type.

If x is the greatest member of T then the value of $\text{succ}(x)$ is undefined.

2.2.5.3 Quantifiers

FDL has two quantifiers: *for_all* and *for_some* as introduced earlier. A quantified-expression consists of the quantifier, the quantified variable (with its associated range of values) and the boolean-valued expression that is being quantified over.

Examples:

$\text{for_all}(x:\text{real}, \text{for_some}(y:\text{real}, x + y = 0));$

2.2.5.4 Sequences

FDL has the following operations and functions for sequences:

Given a sequence $x[a_1, a_2, \dots, a_n]$,

$\text{first}(x)$: denotes the first element of sequence x , which is $[a_1]$

$\text{last}(x)$: denotes the last element of sequence x , which is $[a_n]$

$\text{length}(x)$: denotes the length of sequence x , which is n

$\text{nonfirst}(x)$: denotes the other than the first elements of sequence x , which is $[a_2, a_3, \dots, a_n]$

$\text{nonlast}(x)$: denotes the other than the last elements of sequence x which is $[a_1, a_2, \dots, a_{n-1}]$

$[]$: denotes a empty sequence.

Suppose given another sequence $y[b_1, b_2, \dots, b_n]$, then

$x @ y$: denotes appending sequence x to sequence y , which becomes a new sequence,
 $[a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n]$

2.2.5.5 Sets

FDL supports the following operations and functions for sets:

Syntax	Explanation
$x \setminus / y$	conjunction
$x \setminus \setminus y$	disjunction
$x \setminus y$	lacking, $x-y$
$x \text{ subset_of } y$	$x \leq y$
$x \text{ strict_subset_of } y$	$x < y$

Table 2.3: FDL Sets operations

2.2.5.6 Attributes

Constant attributes

A SPARK constant attribute is replaced by an FDL constant identifier by joining the attribute prefix and attribute identifier together with a double underline character. Similar to constant declarations as noted in section 2.2.4.2, the value of the constant can be determined by a suitable replacement rule in the corresponding FDL rules file.

Examples:

As we have seen in Figures 2.1 and 2.2, Suppose given a declared subtype in SPARK

type Index is range 1..10;

it then is mapped to the following constant declaration in a *.fdl* file:

```
const index__first : integer := pending;
const index__last : integer := pending;
```

Function attributes

Similar to constant attributes, a SPARK function attribute is replaced by an FDL function identifier by joining the attribute prefix and attribute identifier together with a double underline character.

Examples:

FDL expression	SPARK Attributes
colour__pos(red)	Colour 'Pos(Red)
colour__val(blue)	Colour 'Val(Blue)

Table 2.4: FDL Expression for function Attributes

The complete SPARK FDL attributes are referred to “SPARK95 [23], annex K – Language-defined Attributes”.

2.2.6 The FDL rules file

The FDL rules file contains the proof rules generated by the SPARK Examiner, which are illustrated in Figure 2.3. In FDL, rules are grouped into families. FDL rule starts with the rule family declarations in the following form:

```
rule_family family_name :
  EXPRN1 requires TYPES1 &
  EXPRN2 requires TYPES2 &
  ...
  EXPRNn requires TYPESn.
```

There are many predefined rules available covering the usual theorems of mathematics and logic. For instance, rule families may include arith, logic, enumeration and quantify rules etc. As illustrated above, for each rule family declaration, there is a *family_name* immediately following a keyword ‘rule_family’². Moreover, for every subprogram, SPARK will generate additional rules grouped in rule_family to support the proof. The values of *family_name*, *EXPRN*_{*i*} and *TYPES*_{*i*} depend on different rule families. The reserved word ‘requires’ is used to constrain the types of expressions.

For example, in Figure 2.3, SPARK generates a rule family named as swapelement_rules for a subprogram swapelement:

```
rule_family swapelement_rules:
  X requires [X:any] &
  X <= Y requires [X:ire, Y:ire] &
  X >= Y requires [X:ire, Y:ire].
```

FDL rules support the following specific types:

```
i      integer
ir     integer or real
e      enumeration
ire    integer, real or enumeration
any    all types
```

Following the rule family declaration, comes the rule declarations. A rule declaration starts with a unique rule_identifier, following a rule_definition.

Each FDL rule definition supports the two types of rules: *inference rules* and *substitution rules*. Inference rules allow existing facts (i.e. an FDL expression) to be combined to infer new facts. An inference rule definition starts with an FDL expression containing either a

² Although not defined as a reserved word in FDL, ‘rule_family’ should be considered as a reserved word as well. It should be avoided to use as an identifier.

‘may_be_deduced_from’ or a ‘may_be_deduced’ reserved word and an optional *list_conditions* part. The *list_conditions* part starts with an ‘if’ reserved word, following an FDL expression. Substitution rules allow one FDL expression of a verification condition to be replaced by another equivalent FDL expression. A substitution rule definition starts with either an *one_way_rule* or a *two_way_rule* expression. One-way rules allow substitution one way round by using the reserved word ‘may_be_replaced_by’ whereas two-way rules uses the reserved word ‘are_interchangeable’ and allows interchanging either way (if condition(s) applicable).

Example:

```
swapelement_rules(1): character__pos(X) may_be_replaced_by X.
swapelement_rules(2): character__val(X) may_be_replaced_by X.
swapelement_rules(3): integer__first <= integer__last may_be_deduced.
swapelement_rules(4): integer__base__first <= integer__base__last may_be_deduced.
swapelement_rules(5): integer__base__first <= integer__first may_be_deduced.
swapelement_rules(6): integer__base__last >= integer__last may_be_deduced.
swapelement_rules(7): index__first may_be_replaced_by 1.
swapelement_rules(8): index__last may_be_replaced_by 10.
swapelement_rules(9): index__base__first <= index__base__last may_be_deduced.
swapelement_rules(10): index__base__first <= index__first may_be_deduced.
swapelement_rules(11): index__base__last >= index__last may_be_deduced.
```

In the above example copied from Figure 2.3, *swapelement_rules*1, 2, 7 and 8 are one-way substitution rules. All other rules are inference rules. Figure 2.7 illustrates a more complicate example of FDL rules file fragment with optional *list_conditions* parts.

Example:

```
...
insert_rules(37): kernel__task_states__val(X) <= kernel__running may_be_deduced_from
    [0 <= X, X <= 8].
insert_rules(38): succ(kernel__task_states__val(X)) may_be_replaced_by
    kernel__task_states__val(X+1) if [0 <= X, X < 8].
insert_rules(39): pred(kernel__task_states__val(X)) may_be_replaced_by
    kernel__task_states__val(X-1) if [0 < X, X <= 8].
...
```

Figure 2.7 An example of a FDL rules .rls file fragment.

As discussed in section 2.2.4.2 constant declarations and section 2.2.5.6 constant attribute parts, the values of constants generated by the Examiner are stored in the *.r/s* file.

Example:

In SPARK source code:

```
type Index is Integer range 1..10;
```

In FDL a *.fdl* file:

```
const index_first : integer := pending;  
const index_last : integer := pending;
```

Thus, corresponding to the above declarations of constants `index__first` and `index__last` in the *.fdl* file, the pending information is given in the *.r/s* file as:

```
swapelement_rules(7): index__first may_be_replaced_by 1.  
swapelement_rules(8): index__last may_be_replaced_by 10.
```

All other inference rules defined in Figure 2.3 can be easily deduced from the SPARK code. As for the first two function attributes rules that are generated by the Examiner for every *.r/s* file, are just dummy declarations in the *.r/s* file.

2.2.7 The FDL Verification Conditions file

Similar to a mathematical proof, program verification using formal verification methods refers to given the formal specifications of the requirements in a program, where the specifications are expressed as a certain form of precondition and postcondition, the program is correct if it meets the specifications. SPARK applies the well-known Hoare's

partial correctness methodology [24] to verify a program. A program is said to be partially correct if, given that the precondition is satisfied on entry to the program and the program terminates, then the postcondition holds upon exit from the program. In SPARK, preconditions and postconditions are expressed as annotations as described earlier. A verification condition or a theorem is a formula for the correctness of a particular path between two *cutpoints* — the precondition, the postcondition and any other assert annotations that present in the body of the code. The generated VCs are stored in the FDL *.vcg* file. The number of VCs in each *.vcg* file depends on how many paths or cutpoints exist in the source SPARK program.

As illustrated in Figure 2.1 and 2.4, since *swapelement.ada* only contains one postcondition defined in line 12 with no precondition or assert annotations, the generated theorem only has one VC.

Generally, each verification condition generated by the SPARK Examiner has the following form:

VC path identifier

$ \left. \begin{array}{l} H_1: \dots \\ H_2: \dots \\ \dots \\ H_n: \dots \\ \rightarrow \\ C_1: \dots \\ C_2: \dots \\ \dots \\ C_n: \dots \end{array} \right\} $	Verification Condition or Theorem
--	-----------------------------------

A VC path identifier specifies a path number with the associated procedure or function. For instance, ‘procedure_swapelement_1’ is a VC path identifier illustrated in Figure 2.4.

Each H_i is a hypothesis and each C_i is a conclusion. The above form can also be seen as the following form:

$$(H_1 \wedge H_2 \wedge \dots H_n) \rightarrow (C_1 \wedge C_2 \wedge \dots C_n)$$

Each H_i and C_i comprises different FDL expressions. However, if a verification condition or theorem is trivially true, then the above form becomes:

*** true . /* trivially true VC removed by Examiner */

The following example show a *.vcg* file containing three VCs:

```

1  package P
2  is
3  end P;
4
5  package body P
6  is
7      procedure IntRoot(N : in Natural;
8          Root : out Natural)
9          --# derives Root from N;
10         --# pre N >= 0;
11         --# post (Root * Root <= N) and
12         --# (N < (Root + 1) * (Root + 1));
13         is
14             R : Natural := 0;
15             S, T : Natural := 1;
16         begin
17             loop
18                 --# assert (T = 2 * R + 1) and
19                 --# (S = (R + 1) * (R + 1)) and
20                 --# (R * R <= N);
21                 exit when S > N;
22
23                 R := R + 1;
24                 T := T + 2;
25                 S := S + T;
26             end loop;
27             Root := R;
28         end IntRoot;
29 end P;
```

Figure 2.8 A simple example of SPARK program IntRoot.ad

The *IntRoot* routine in Figure 2.8 simply finds the integer square root of a non-negative number by starting the search at zero and trying successive candidates in turn until the root is found. Running the Examiner produces the following VCs:

For path(s) from start to assertion of line 18:

```

procedure_introot_1.
H1: n >= 0 .
H2: n >= natural__first .
H3: n <= natural__last .
->
C1: 1 = 2 * 0 + 1 .
C2: 1 = (0 + 1) * (0 + 1) .
C3: 0 * 0 <= n .

```

For path(s) from assertion of line 18 to assertion of line 18:

```

procedure_introot_2.
H1: t = 2 * r + 1 .
H2: s = (r + 1) * (r + 1) .
H3: r * r <= n .
H4: not (s > n) .
->
C1: t + 2 = 2 * (r + 1) + 1 .
C2: s + (t + 2) = (r + 1 + 1) * (r + 1 + 1) .
C3: (r + 1) * (r + 1) <= n .

```

For path(s) from assertion of line 18 to finish:

```

procedure_introot_3.
H1: t = 2 * r + 1 .
H2: s = (r + 1) * (r + 1) .
H3: r * r <= n .
H4: s > n .
->
C1: r * r <= n .
C2: n < (r + 1) * (r + 1) .

```

Figure 2.9 A simple example of SPARK FDL .vcg file with 3 VCs.

In Figure 2.9, VCs are generated from the start of the subprogram to the assert statement (loop invariant); from the assert statement round the loop back to the assert statement; and from the assert statement to the subprogram end. In addition, an FDL .vcg file may have optional traceability comments. For example, ‘*For path(s) from start to assertion of line 18:*’ is such a traceability comment. Traceability comments state what the ‘kinds’ of VCs are and which portions of the SPARK code they are generated from.

Chapter 3 The PVS Language

3.1 PVS Overview

Prototype Verification System (PVS) is a comprehensive formal specification and verification system developed by the SRI International Institute [26]. Similar to SPARK, PVS also includes a specification language, namely *PVS language* and an automatic verification tool, namely *PVS theorem prover*. However, unlike SPARK FDL, which is based on the first-order logic technique, PVS language is built on more rich Higher-Order Logic [10, 25]; i.e. not only can PVS deal with boolean expressions, binding information similar to the first-order logic, but also functions in PVS can take functions as arguments and return them as values. In addition, quantifiers in PVS can be applied to function variables as well [8]. PVS specifications are organized into parameterized theories that may contain definitions, axioms, and theorems. The PVS theorem prover has primitive inference procedures that are applied interactively under user guidance. The inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic [8].

In this chapter, only relevant aspects of the PVS language will be described, and they are referenced from the PVS Language Reference manual [27]. For the full descriptions of the PVS language and integrated verification tool, please refer to [8, 27] and the PVS website at <http://pvs.csl.sri.com/>.

3.2 The PVS Language

3.2.1 Identifiers, reserved words and comments

PVS identifiers are case sensitive, thus 'FOO', 'Foo' and 'foo' are different identifiers. However, the reserved words in PVS are not case sensitive, but in convention, those reserved words are displayed as upper case in order to distinguish them from identifiers. For instance, logic operations 'AND', 'IMPLIES', 'OR', binding expressions 'EXISTS', 'FORALL', and 'AXIOM' are all PVS reserved words. The full list of PVS reserved words can be found in Appendix B.

In PVS, a comment begins with a '%' character followed by any sequence of characters and terminated by a newline.

Example:

```
% This is a PVS comments.
```

3.2.2 Declarations

Similar to SPARK, PVS declarations specify types, constants, variables, as well as other type systems. In addition, PVS declarations contain formula declarations and various other definitions that beyond the scope of this thesis.

3.2.2.1 Type Declarations

PVS has three basic kinds of type declarations: *uninterpreted type*, *interpreted type* and *enumeration*

type. In addition, PVS can build more complex type expressions by using *type constructors*. There are type constructors for *function types* and *record types* etc.

The uninterpreted type

The uninterpreted type simply introduces a type with no constraints on an implementation of the specification.

Example: `t1, t2: TYPE`

Further constraints may be provided by formula declarations starts with the reserved word `AXIOM` (see Section 3.2.2.4).

The interpreted type

The interpreted type specifies a concrete type for a type declaration.

Example:

`fun: TYPE = [real --> int]`

introduces a concrete type. Its name is 'fun' for the type of functions with domain from real to integer.

The enumeration type

Similar to SPARK FDL, the enumeration type has the following form:

`enum: TYPE = {e1, ..., en};`

Example:

`Colour: TYPE = {red, orange, yellow, green, blue, indigo, violet};`

Function Type Declarations

In PVS, a function type declaration has three equivalent forms:

- $[t_1, \dots, t_n \rightarrow t]$
- `FUNCTION` $[t_1, \dots, t_n \rightarrow t]$
- `ARRAY` $[t_1, \dots, t_n \rightarrow t]$

Where each t_i is a type expression. Thus, in PVS, an array is also treated as a function type.

Examples:

`AND: [bool, bool -> bool];`

A boolean function 'AND' has two boolean elements and whose range is a boolean as well.

Record Type

Record Type has the following form:

$[\# a_1: t_1, \dots, a_n: t_n \#]$

a_i denotes record fields and t_n denotes record types.

Examples:

```
daynumber : TYPE = int;
monthname : TYPE = {jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
yearnumber : TYPE = int;
date : TYPE = [# day: daynumber, month : monthname, year : yearnumber #];
```

3.2.2.2 Constant Declarations

There are two kinds of constants in PVS: uninterpreted and interpreted constants. An uninterpreted constant only specifies the type with no value but an interpreted constant specifies a value for the constant.

Example:

```
j: int;  
t: bool = TRUE;
```

3.2.2.3 Variable Declaration

A variable declaration introduces a new variable with an associated type. The variables in PVS are simply logical variables, not program variables. Thus, PVS variables are usually used in binding expressions such as EXISTS, FORALL and LAMBDA.

Example:

```
f: THEOREM (FORALL (x: nat): x>0)
```

The occurrence of variable x in a theorem f is of natural integer type.

3.2.2.4 Formula Declarations

Formula declarations start with an identifier associated with axioms or theorems declarations.

The body of formula is comprised of boolean expressions.

Examples:

```
real_nonempty: AXIOM (EXISTS(x: real): TRUE);  
excluded_middle: THEOREM (FORALL(a: bool): OR(a, NOT(a)));
```

3.2.3 PVS Expressions

Similar to other languages, PVS expressions also consist of arithmetic and logical operators, function applications, quantifiers etc. The PVS system has a *prelude library* containing all definitions of operators and their precedences. The prelude library also contains many

fundamental mathematical theories (such as booleans, integers, reals etc) available to the PVS theorem prover for the further reasoning. Moreover, PVS expressions support both prefix and infix forms. In other words, $x+1$ and $+(x, 1)$ are semantically equivalent. PVS uses ‘;’ as a separator to terminate a specification.

3.2.3.1 Boolean expressions

The Boolean expressions include the constants TRUE and FALSE, the unary operator NOT, and the binary operators AND (also written &), OR, IMPLIES ($\Rightarrow$), WHEN, and IFF ($\Leftrightarrow$). The declarations for these are specified in the *boolean prelude theory*. All of these have their standard meaning, except for WHEN, which is the converse of IMPLIES (i.e. $A \text{ WHEN } B \equiv B \text{ IMPLIES } A$).

3.2.3.2 Numeric expressions

Similar to other languages, the numeric expressions include the numerals, the unary operator -, the binary operators +, -, *, and /, and relational operators <, <=, > and >=.

3.2.3.3 Function expressions

PVS function expressions are same as those of other language. A function f to expression x is simply denoted as $f(x)$.

3.2.3.4 Binding expressions

PVS binding expressions were actually introduced before. Here, they can be summarized as

follows: PVS binding expressions include two quantifiers: **FORALL** and **EXISTS** and a **LAMBDA** expression. Thus, the bindings specify the variables bound by the operators.

Examples:

FORALL((x:int, y:int): x + y = 0);

EXISTS(x:real: x > 0);

LAMBDA(t:bool : t = TRUE);

3.2.3.5 Record Expressions

Record expressions are of the form $(\# a_1 := e_1, \dots, a_n := e_n \#)$, which has type $[\# a_1 : t_1, \dots, a_n : t_n \#]$, where each e_i is of type t_i .

Example:

date : ($\#$ day := int, month := monthname, year := int $\#$);

3.2.3.6 Record Accessors

PVS uses ‘ $\cdot$ ’ to access the field of each record type. For instance, if r is of type

$[\# x, y : \text{real} \#]$, then the x field can be accessed using the expression $r \cdot x$.

3.2.3.7 Override Expressions

Functions and records may be updated by the override expressions. Of course, the result of an override expression is also a function and record that is exactly the same as the original, except that at the specified arguments it takes the new values.

Example:

Given a function $f1: \text{ARRAY}[1, 2, \dots, i, i+1, \dots, n]$, $f1$ can be overridden by modifying i^{th} element as j , expressed as: $f1 \text{ WITH } [(i) := j]$;

3.3 The PVS specification file

As noted in section 3.1, PVS specifications consist of a collection of theories. A theory has a *theory identifier*, a *theory body* and an *ending id*. A theory body includes a list of declarations, axioms and various theorems if applicable. The PVS specification files have the .pvs extension. For example, Figure 3.1 illustrates an example of PVS specification:

```
swapelement: THEORY
BEGIN

%% % define fdl mod function using PVS rem function
mod: [int,int -> int] = (lambda(x: int, y: int): rem(abs(x),abs(y)));
%% % procedure swapelement
round__: [real -> int];
atype: TYPE = array[int -> integer];
index: TYPE = int;
index__last: int;
index__first: int;
integer__last: int;
integer__first: int;
%% %
%% %
%% %
%% % rule_family swapelement_rules:
swapelement_rules(3): AXIOM <=(integer__first, integer__last);
swapelement_rules(7): AXIOM =(index__first,1);
swapelement_rules(8): AXIOM =(index__last,10);
procedure swapelement_1: THEOREM (forall(i: int, j: int, a: atype): ) AND (TRUE, >= (i,
index__first),<=(i,index__last),>=(j,index__first),<=(j,index__last), (forall(i__1: index):
IMPLIES (AND(>=(i__1, index__first), <=(i__1, index__last)), AND(>= (a(i__1),
integer__first),<=(a(i__1),integer__last))))IMPLIES(AND(=((a WITH [ (i) := a(j)]) WITH
[ (j) := a(i)]),(a WITH [ (i) := a(j)]) WITH [ (j) := a(i)])))));

END swapelement
```

Figure 3.1 An example of PVS specification file.

Chapter 4 OpenMath

4.1 Overview

OpenMath is a project to propose the development of a series of standards for the semantically rich representation of mathematical objects. It was originally advocated by European mathematicians, but now an international OpenMath Consortium has been formed. The consortium proposes a standard to represent mathematical objects electronically. The motivation is very straightforward. Mathematical information is in poor relation with electronic publishing, and mathematical objects communicate ineffectively among computer programs. There is strong need to create “a standard for representing mathematical objects with their semantics, allowing them to be exchanged among computer programs, stored in databases, or published on the worldwide web”[22]. Like the Mathematics Markup Language (MathML)[11], which was the first Worldwide Web Consortium (W3C) [12] recommendation for the Extensible Markup Language (XML)[13], OpenMath is also a XML application. However, unlike MathML, which primarily deals with display or rendering of mathematical objects, OpenMath tightly focuses on mathematical semantic meaning or contents. OpenMath inherits intrinsic characteristics of the XML language. Mathematical objects encoded in OpenMath can be displayed in a browser, exchanged among software systems, stored and retrieved from databases or files and even verified as being mathematically sound or not. Although MathML and OpenMath are highly complementary to each other, we are more interested in the semantic meaning or content of

mathematical objects rather than the presentation of mathematical expressions. Therefore, this chapter mainly describes the OpenMath objects and encoding mechanism. As for MathML, interested reader can refer to www.w3.org/math for more detailed information. A brief comparison of OpenMath and MathML can be found in [33].

4.2 The OpenMath Objects and Encodings

4.2.1 Basic OpenMath Objects and Encodings

The OpenMath representation of a mathematical structure is referred to as an *OpenMath object*. Formally, an OpenMath object or OpenMath expression is a labeled tree, whose leaves are the *basic OpenMath objects*, which are encoded within pre-defined XML tags. The basic OpenMath objects include: *integers*, *IEEE double precision floats*, *Unicode strings*, *byte arrays*, *variables* or *symbols* [34]. Those basic OpenMath objects are encoded as the following XML element tags respectively:

integer: <OMI>
float: <OMF>
string: <OMSTR>
bytearray: <OMB>
variable: <OMV>
symbol: <OMS>

Of these, <OMI>, <OMSTR> and <OMB> are simple elements with no attributes. For instance, OpenMath encodes the integer 2 as: <OMI>2<OMI>. <OMF> and <OMV> both have one compulsory attribute. <OMF> has either the attribute *dec* or the attribute *hex*.

For example, `<OMF dec="1.0e-10"/>` is a valid expression for a float number. Of course, the two attributes cannot be present simultaneously. `<OMV>` also has only one attribute *name*, whose value is the variable name. For instance, a variable x encodes as: `<OMV name="x" />`.

Perhaps the most interesting element is the *symbol* `<OMS>` element. It carries two compulsory attributes *cd* and *name*. *cd* represents *Content Dictionary* (CD in short, and it will be covered in detail in the next section), an external OpenMath document that defines the meanings of OpenMath objects being transmitted. OpenMath uses the CD as a mechanism to define the numerous rich mathematical concepts or objects. The value of *cd* is a name of the Content Dictionary in which the symbol is defined and the value of *name* gives the name of the symbol [34]. For example, `<OMS name="plus" cd="arith1"/>` represents the usual *plus* function, as defined in the CD "arith1". OpenMath uses `<OMOBJ>` denoting an OpenMath object. Therefore, a basic OpenMath object can be simply encoded as:

```
<OMOBJ>
  <OMS name="sin" cd="transc1"/>
</OMOBJ>
```

4.2.2 Compound OpenMath Objects and Encodings

Except the basic OpenMath objects discussed earlier, OpenMath provides the following encodings that can recursively build up *compound OpenMath objects*:

```
application: <OMA>
attribution: <OMATTR> and <OMATP>
binding: <OMBIND> and using <OMBVAR> to tag the bound variables
```

Unlike the basic OpenMath objects, the compound OpenMath objects have no attribute(s),

but they all can include either the basic OpenMath objects or compound OpenMath objects recursively to construct a tree-like OpenMath object representation.

For example, an expression $\sin(x)$, the simplest function application, can be represented by the compound OpenMath objects encoding as follows:

```
<OMOBJ>
  <OMA>
    <OMS name="sin" cd="transc1"/>
    <OMV name="x"/>
  </OMA>
</OMOBJ>
```

Another more complicated example is binding objects that are used to represent an expression containing bound variables, for example:

```
<OMOBJ>
  <OMBIND>
    <OMS cd="quant1" name="forall"/>
    <OMBVAR>
      <OMV name="x"/>
    </OMBVAR>
    <OMA>
      <OMS name="sin" cd="transc1"/>
      <OMV name="x"/>
    </OMA>
  </OMBIND>
</OMOBJ>
```

represents $\text{forall}(x), \sin(x)$.

Attribution is often encoded by using `<OMATTR>` and `<OMATP>` together as a pair to denote the type of a variable. The following example is an OpenMath attribution object encoding used to express the variable x that is attributed with a type for *real* numbers.

Example:

```
<OMOBJ>
  <OMATTR>
    <OMATP>
      <OMS cd="mathmltypes" name="type" />
      <OMS cd="mathmltypes" name="real_type" />
    </OMATP>
    <OMV name="x" />
  </OMATTR>
</OMOBJ>
```

4.3 Content Dictionaries

From above, we have seen that OpenMath actively builds on an extremely simple kernel. Recall from the preceding section, OpenMath uses a mechanism called *Content Dictionary* to fully represent the numerous rich mathematical objects. CDs are central to the OpenMath philosophy of transmitting mathematical information [34]. CDs are used to assign informal and formal semantics to all symbols used in OpenMath objects [34]. They define the meaning of symbols used to represent the mathematical concepts expressed by OpenMath objects. CDs use two elements `<CMP>` and `<FMP>` to define a symbol's properties informally and formally:

- `<CMP>`: Comments, a set of a symbol's properties in plain text (*Commented Mathematical Properties*)
- `<FMP>`: Properties, a set of a symbol's properties encoded in OpenMath (*Formal Mathematical Properties*)

The following example defines a binary commutative function *plus*:

for all a, b; $a + b = b + a$

```
<CMP> a + b = b + a </CMP>
<FMP>
<OMOBJ>
  <OMBIND>
    <OMS cd="quant1" name="forall"/>
    <OMBVAR>
      <OMV name="a"/>
      <OMV name="b"/>
    </OMBVAR>
    <OMA>
      <OMS cd="relation1" name="eq"/>
      <OMA>
        <OMS cd="arith1" name="plus"/>
        <OMV name="a"/>
        <OMV name="b"/>
      </OMA>
      <OMA>
        <OMS cd="arith1" name="plus"/>
        <OMV name="b"/>
        <OMV name="a"/>
      </OMA>
    </OMA>
  </OMBIND>
</OMOBJ>
</FMP>
```

For convenience, CDs are divided into four categories and are grouped into different groups, called *CD Groups*. The mechanisms of CD Groups allow people or applications to identify the collections of CDs easily. However, CD Groups do not affect the OpenMath objects themselves. Symbols in an OpenMath object always refer to the CDs, not the CD Groups.

The four categories are the *core*, *public*, *extra* and *contributed* CDs. The core and public categories are identified as OpenMath’s “official” CDs, because these CDs have been reviewed and endorsed by the OpenMath society. For instance, the core category contains the most common set of CD Groups – MathML CD Group, which is designed to ensure compatibility with the Content Markup of MathML 2 [35]. The MathML Group contains the so-called K-12 (from Kindergarten to high school 12th grade) fragment of mathematics concepts, such as algebra, arithmetic, function, geometry, integer, logic, set, quantity, vector etc. Due to the natural evolution of mathematics, the CD Groups are growing constantly. Currently, OpenMath has two categories: extra and contributed CDs. These two categories include linear algebra, polynomials and group theory etc. but they are not encoded in this thesis. All OpenMath objects encoded in this thesis are from either core or public categories. Reader please refer to the <http://www.openmath.org/> website for a complete list of CDs. In addition, OpenMath does not mandate applications using only the “official” set of CDs and it is often useful to design “private” CDs for a specific purpose. As an extension of OpenMath, OMDoc, which will be discussed in the next chapter, defines many CDs for its own purpose.

Chapter 5 OMDoc

5.1 Overview

As introduced in the last chapter, as a mathematical markup language, OpenMath mainly dealt with mathematical objects and formulae, where the meaning of mathematical concepts and knowledge were relatively fixed. As an extensible library mechanism, Content Dictionaries, instead of being mathematical objects, were primarily served as background references defining the meaning of symbols in OpenMath [32]. In addition, OpenMath did not provide markup for mathematic forms like theories, axioms, definitions, theorems, and proofs. However, in formal program verification area, this level of support is crucial. As illustrated in Chapter 2, the outputs of SPARK in the FDL modeling language form are comprised of various declarations(definitions), rules(axioms), verification conditions (theorems).

Fortunately, OMDoc [14], another content-based XML markup scheme, from the document and theory level supports formal markup for mathematical knowledge. OMDoc stands for *Open Markup of Mathematical Documents*. It can be considered as an extension for the OpenMath proposal. As an extension of OpenMath, OMDoc inherits all OpenMath's mechanisms. In addition, in the theory level, OMDoc adds markup for mathematic concepts like theories, axioms, definitions, theorems, and proofs. From the document level, OMDoc also extends the OpenMath CDs mechanism by adding a mechanism so that the OpenMath *symbol* <OMS> elements can be “self-contained” in the inter-document. Also, OMDoc

provides another mechanism for referencing and locating external OMDoc documents. More ambitiously, The OMDoc proposal attempts to provide a universal content-based markup language (called an interlingua in OMDoc) that covers all aspects of mathematical concepts, and allows the development of semantics-based value-added services for displaying and manipulating mathematical knowledge among mathematical-based software systems [14].

To make more practical use of the OMDoc format as an interlingua, OMDoc has even provided several interfaces that transform the OMDoc documents into other formats [14, 28, 36]. As OMDoc is an XML, these interfaces, which are in the form of the Extensible Stylesheet Language Transformation (XSLT) [19] formats, can perform two levels transformation from the OMDoc documents. One is the presentation level; XSLT style sheets can transform OMDoc documents in HTML, LaTeX and MathML formats for human readers, and another is the machine consumption level (particularly for theorem proving systems). XSLT style sheets can transform OMDoc documents into several theorem prover language syntaxes (including PVS language syntax). So it is possible to achieve the interoperability for n systems with only $2n$ translations instead of n^2 [14].

OMDoc is a large project, and it contains enormous special XML elements and attributes (the so-called OMDoc elements and attributes) for mathematical proofs, abstract data types, complex theories and inheritances, which are not our concerns in this thesis. In this chapter, only relevant OMDoc elements and attributes will be introduced and they are referenced from [14]. For complete information about the OMDoc project, please refer to [14] and [36].

5.2 OMDoc Statements

In OMDoc, mathematical forms like axioms, definitions and theorems are called *mathematical statements*, since they state properties about mathematical objects. Axioms and definitions hold the meaning of symbols they define. Once defined, those symbols can be used to build up other mathematical objects. Theorems state properties about mathematical objects, which can be proven from the axioms and definitions. OMDoc mathematical statements discussed in this thesis are comprised of the following elements: *symbol*, *type*, *axiom*, *definition* and *assertion*. The following table lists relevant OMDoc mathematical statement elements, their attributes and the contents (children). For convenience, it also illustrates the parent of these mathematical statement elements – *theory* element. This section discusses mathematical statement elements and their attributes. *Theory* element will be covered in the next section.

Element	Attributes			Content
	Required	Optional	Optional value	CMP*, type*
symbol	id	kind	type, object	
		scope	global, local	
type	system	id, for, style		CMP*, OMOBJ
axiom	id			CMP*, FMP*
definition	id, for	type	simple	CMP*, FMP*
assertion	id	type	theorem	CMP*, FMP*
theory	id			CMP*, statement*

Table 5.1: Symbol, definition, axiom, assertion and theory in OMDoc.

As illustrated from Table 5.1, almost all of these elements have their common parts for specifying mathematical properties. These common parts include <CMP> or <FMP> elements.

The <CMP> and <FMP> elements play the same roles as defined in OpenMath. That is: <CMP> specifies the informal mathematical concepts and <FMP> describes the formal

meaning of mathematical concepts. However, unlike in OpenMath, `<CMP>` can only contain plain text to declare a symbol's properties, `<CMP>` in OMDoc may also include arbitrary text interspersed with `<OMOBJ>` encoding element.

5.2.1 Symbol Declaration

The *symbol* element specifies a symbol of a mathematical concept, such as 1 for the natural number *one* and + for *addition*. The *symbol* element has a required attribute *id* and optional attributes *kind* and *scope* etc. The *id* attribute uniquely identifies the symbol itself. The attribute *kind* can take a value such as 'type', which denotes a type system (PVS for instance), or 'object', which is default for any other symbols. The value of *scope* attribute is either 'global' or 'local'; if it is 'local', then the symbol cannot be exported outside the theory.

5.2.2 Type Declaration

In OMDoc, as a child element of a *symbol* element, the *type* element specifies type information for the symbol it is contained in. The value of the *system* attribute of *type* element names the type of system. Figure 5.1 illustrates an example of a symbol *f* and its child type declaration.

```
<symbol id="f">
  <type system="sts">
    <OMOBJ>
      <OMA>
        <OMS cd="mathmltypes" name="fn_type" />
        <OMS cd="mathmltypes" name="real_type"/>
        <OMS cd="mathmltypes" name="integer_type"/>
      </OMA>
    </OMOBJ>
  </type>
</symbol>
```

Figure 5.1 An example of a symbol and a type declaration in OMDoc.

In Figure 5.1, a *symbol* element is declared by using *id* attribute that takes *f* as its value. Then in a standard OpenMath CD, denoted as *Small Type System* or ‘sts’ as the *system* attribute, *f* is specified as a function type and takes argument real type and returns an integer type by using OpenMath standard *mathmltype* CD. Thus, the OMDoc fragment in Figure 5.1 represents: $f(\text{real}) \rightarrow \text{integer}$.

5.2.3 Definition

The *definition* element is used to fix the meaning of a symbol in terms of already defined one. Such symbols are called *defined* symbols, otherwise they are *primitive*’s. The *definition* element has two compulsory attributes *id* and *for* to fix the meaning of a symbol that is defined. The optional *type* attribute with “simple” value refers to an OpenMath object that can be substituted for the symbol specified in the *for* attribute of the definition.

Example:

Suppose given an identifier *index* and its type is an *integer*, declared as:

type index = integer;

It can be expressed as a symbol and definition declarations in Figure 5.2 as follows:

```
<symbol id="index" kind="type" scope="global">
  <type system="sts">
    <OMOBJ>
      <OMS cd="sts" name="type" />
    </OMOBJ>
  </type>
</symbol>
<definition id="index" for="index" type="simple">
  <FMP>
    <OMOBJ>
      <OMS cd="mathmltypes" name="integer_type"/>
    </OMOBJ>
  </FMP>
</definition>
```

Figure 5.2 An example of a symbol and its definition in OMDoc.

It is worth pointing out that by nature, a *definition* element should immediately succeed its defined *symbol* element. The relationship of a *definition* element and a *symbol* element would be siblings to each other.

5.2.4 Axiom

Similar to the *definition* element, the *axiom* element can also define the meaning of a symbol. However, unlike the *definition* element that gives the “fixed” meaning of a *defined* symbol, the *axiom* element specifies the “constrained” meaning of a *primitive* symbol. The *axiom* element declaration in OMDoc is simpler than the *definition* element declaration. It only has one required attribute *id*. The *axiom* element allows the expressing the mathematical content of axioms or logic formulas.

Example:

```
<symbol id="real-non-empty" >
  <CMP>The type real is non-empty
</CMP>
</symbol >
.....
<axiom id="real_nonempty" >
  <FMP>
    <OMOBJ>
      <OMBIND>
        <OMS cd="sts" name="exists"/>
        <OMBVAR>
          <OMATTR>
            <OMATP>
              <OMS cd="sts" name="type"/>
              <OMS cd="reals" name="real"/>
            </OMATP>
            <OMV name="x"/>
          </OMATTR>
        </OMBVAR>
        <OMS cd="booleans" name="TRUE"/>
      </OMBIND>
    </OMOBJ>
  </FMP>
</axiom>
```

Figure 5.3 An example of a primitive symbol and its axiom definition in OMDoc.

The OMDoc fragment in Figure 5.3 may denote the following axiom or formula:

`real_nonempty: AXIOM (exists(x: real): TRUE).`

5.2.5 Assertion

OMDoc uses the *assertion* element for all statements (proven or not) about mathematical objects. The *assertion* element also has a required *id* attribute, which allows the referencing of it in a document, and an optional *type* attribute, which can have a value, such as ‘theorem’. Figure 5.4 gives the simplest example of an assertion, i.e. a theorem simply contains a “true” statement.

```
<assertion id="f" type="theorem">
  <FMP>
    <OMOBJ>
      <OMS cd="logic1" name="true" />
    </OMOBJ>
  </FMP>
</assertion>
```

Figure 5.4 A TRUE assertion declaration in OMDoc.

5.3 Theory

In OMDoc, all mathematical statements, including symbols, definitions, axioms and assertions, are used together to construct a theory. A theory serves as a framework for semantically referencing mathematical concepts.

OMDoc uses the *theory* element to specify a theory. The *theory* element has a required *id* attribute for referencing the theory. The contents or children of the *theory* element may include various *symbol*, *definition*, *axiom* and *assertion* elements. If any of such information is

changed, it will yield a different theory. Therefore, the *theory* element must be a parent of other statements. Similar to the relationship between the *symbol* and *definition* elements, the relationship among these *theory* children elements would be siblings to each other.

In addition, in OMDoc, theories serve another purpose. From above example we have seen, as an extension of OpenMath, OMDoc documents contain definitions of mathematical concepts, which need to be referred using OpenMath encodings. In particular, documents describing theories even reference themselves. OMDoc theories adapt the OpenMath Content Dictionary mechanism by allowing them to be referenced by using `<OMS>` CD's attributes in the *theory* elements. In this case, the value of the *id* attribute of the *theory* element can substitute the value of *cd* attribute of the `<OMS>` element defined in the OpenMath CD. This mechanism makes the OMDoc format self-contained, as OMDoc documents contain symbols that are declared in the documents themselves.

Example:

Suppose given a PVS theory:

```
if_def: THEORY
BEGIN
  IF: [boolean,T -> T];
END if_def
```

It can be expressed by the following OMDoc form:

```
<theory id=" if_def ">
...
<symbol id="IF" scope="global" kind="object">
  <type system="pvs">
    <OMOBJ>
      <OMA>
        <OMS cd="pvs" name="funtype"/>
        <OMS cd="booleans" name="boolean"/>
        <OMS cd="if_def" name="T"/>
        <OMS cd="if_def" name="T"/>
      </OMA>
    </OMOBJ>
  </type>
</symbol>
...
</theory>
```

Figure 5.5 An OMDoc fragment theory definition.

The *cd* attribute value "if_def" in Figure 5.5 is referred by the theory *id* attribute, instead of the standard OpenMath CDs. Identifying and referencing OMDoc elements and documents are discussed in the following section.

5.4 OMDoc Document and locating

As introduced, OMDoc is an XML application. Thus, OMDoc has only one root element, namely the *omdoc* element. The *omdoc* element contains all other elements described in the earlier in this chapter. The *omdoc* element has a required attribute *id* that can be used to reference the whole document.

In OMDoc, an element can be called a *top-level* element if it can be appeared as a direct child of the *omdoc* element. The *theory* element is such a top-level element, and the *omtext* element is another top-level element.

OMDoc uses the *omtext* element to specify further explanation of mathematical knowledge, such as annotations, remarks and comments. The *omtext* element has an *id* attribute for the reference prupose. The *omtext* element can also contain <CMP> and <FMP> elements. This element should be used if one wants to comment the OMDoc representation. It differs with an XML comment (i.e. anything between “<!--” and “-->”), which is not even read by an XML parser.

As described earlier, OpenMath uses CDs as an extension to reference mathematical objects, and OMDoc adapts the OpenMath CD mechanism to self-contain theories as CD’s

attributes. A question might arise, how can all of these <OMS> attributes (both standard and self-contain attributes) be located? OMDoc contains a repository (or a library) to support all “standard” OpenMath CDs. In addition, OMDoc supports a catalogue mechanism that allows the specifying of the location of defined OMDocs. The specification of a catalogue is done by the *catalogue* attribute in the *omdoc* element. It is a Uniform Resource Identifiers (URI) reference to another OMDoc document whose catalogue is inherited by the referencing one.

By convention, to distinguish from an XML file, an OMDoc document has an extension ‘.omdoc’, but technically, there is no such restriction. The following example illustrates an **if_def.omdoc** OMDoc document:

```
<?xml version="1.0" encoding="UTF-8"?>
<omdoc id="if_def.omdoc" catalogue="catalogue.omdoc">
  <omtext>
    <CMP>if_def theory</CMP>
  </omtext>
  <theory id=" if_def ">
    <symbol id="IF" scope="global" kind="object">
      <type system="pvs">
        <OMOBJ>
          <OMA>
            <OMS cd="pvs" name="funtype"/>
            <OMS cd="booleans" name="boolean"/>
            <OMS cd="if_def" name="T"/>
            <OMS cd="if_def" name="T"/>
          </OMA>
        </OMOBJ>
      </type>
    </symbol>
    ...
  </theory>
</omdoc>
```

5.5 Transforming OMDoc

As XML applications, OMDoc documents can be processed by XSLT style sheets to transform them into other formats. One of the primary goals of the OpenMath and OMDoc is to provide a universal document and language for both human readers and mathematical software systems. To achieve this, OMDoc has provided several XSLT style sheets for transforming OMDoc for presentation (such as in HTML, LaTeX for human readers) and for mathematical software systems (especially for theorem proving systems, such as PVS, TPS, INKA) consumption [36]. It is not our concern in this thesis to discuss the first kind of presentation-based style sheets, but the latter kind of machine-based style sheets will be explained in the next chapter. Transforming OMDoc will be discussed in details by using concrete case study (from transforming OMDoc into PVS language syntax).

Chapter 6 The Translation

6.1 Overview

As discussed in the first Chapter, one of the practical problems that remain to be addressed among different theorem proving systems is the integration issue. One way to solve this problem is from the semantic level to provide translators for specification languages between these systems. In terms of the translation from one language to another, the most obvious way is to transform directly from the source language into the target language. Although this approach seems very straightforward, it has a significant drawback. The drawback of this approach is that the translation is one to one based translation. Therefore, if there are n target languages that need to be translated from one source language, the translations are needed to be done n times. More seriously, if there are n languages that are need to be communicated; the translations are n^2 times. While it may be possible, it would not be feasible to implement such translations.

As we have seen from previous chapters, one of the primary goals of OpenMath and OMDoc is to provide a universal language for mathematical software systems. To accomplish this goal, OMDoc has even claimed to supply a universally accepted standard format (or an interlingua), which is called the *OMDoc document* in the form of the standard XML application. So ideally, the interoperability for n systems can be achieved with only $2n$ translations instead of n^2 [14]. Although it does not seem very straightforward when compared with the first direct translation approach, the OMDoc approach has many advantages due to the benefits of the OMDoc contributions. To translation from one source

language into n target languages via OMDoc approach, the translation from the source language into the OMDoc form needs only be done once. From a software engineering perspective, mathematical theories in the form of OMDoc format provide a consistent structure and make it possible for them to be modularized and reused among different theorem proving systems that can communicate with the OMDoc documents. In addition, to make practical use of the OMDoc format as an interlingua, OMDoc has even provided several interfaces (XSLT style sheets and libraries) that can transform the OMDoc form into designated formats.

In this thesis, as a case study of the OMDoc approach, we have implemented an experimental translator from one formal verification language SPARK FDL into another formal verification language PVS via the OMDoc methodology. Please recall the first chapter for our motivations and reasons of our selections of candidate languages. OMDoc has claimed it contains an interface that can transform the OMDoc form into the PVS syntax. Recall from Chapter 3, The PVS system has a *prelude library* containing all pre-defined operators and their precedences as well as many fundamental mathematical theories (such as booleans, integers, reals etc). The prelude library can be referred directly by user-defined specifications (in the format of PVS language syntax) to construct various theories. This means that users do not need to create those fundamental mathematical theories anymore. The interface provided by OMDoc not only includes an XSLT style sheet, namely *omdoc2pvs.xsl*, but also includes a corresponding PVS prelude library in the form of the OMDoc document. Such an interface can transform an OMDoc document into the PVS language syntax.

Based on the OMDoc methodology, ideally, the translation from the SPARK FDL syntax into the PVS syntax involves two steps: the first step is to translate SPARK FDL into an OMDoc document and the second step is to apply the PVS-based interface provided by

OMDoc to transform the OMDoc document into the PVS language syntax.

As addressed before, SPARK FDL is based on the relatively simple first-order logic. The PVS language is built on a more powerful higher-order logic containing the first-order logic concepts. In addition, as mentioned above, OMDoc has already provided the interface from OMDoc to PVS. Thus, one would think the implementation of the translation from the simple FDL first-order logic to the more rich PVS higher-order logic via the OMDoc approach should be relatively easy as well.

But in practice, development of the translator has demonstrated that the translation is not as easy and straightforward as we expect. Of course, there are a lot of details that are needed to be considered while translating two different languages. In addition, there are some unexpected problems that emerged during the translator development.

First, most difficulties are caused by the SPARK FDL language itself. The SPARK FDL specifications are generated by the SPARK tool and saved in three separate files respectively.

Recall from Chapter 2, in the FDL declarations file, the “pending” information of constant declarations is actually explicitly explained in the FDL rules file. In the FDL rules file, some rules generated by the SPARK tool contain built-in SPARK attributes (such as `character__pos` and `character__val`) that are not even present in the FDL declaration files. In addition, in the FDL verification conditions file, some expressions (such as the record type) contain extra prefixes (e.g. `fld_` and `upf_`) that are also not declared in the declaration section. All of these expressions are SPARK FDL exclusive and other systems cannot handle them properly. Thus, they cannot be directly translated into the OMDoc document.

Secondly, since the translation is based on the OMDoc approach, the translation must

conform to the OMDoc format. Although OpenMath and OMDoc can express numerous rich mathematical symbols and concepts, the Content Dictionaries in OpenMath and OMDoc only support simple types such as integer, real, boolean etc. But they do not have expressions for data types, such as array, enumeration and record. These data types are fundamental in many verification languages including SPARK FDL and PVS. Therefore, some ‘private’ CDs have to be specified to support these data types.

Even more surprisingly, instead of using the standard OpenMath CDs, the interface (including both the XSLT style sheet and the prelude library) for PVS provided by OMDoc uses specific variant CDs. Not to mention documents containing private CDs, the OMDoc document containing only the “standard” CDs itself cannot even be transformed correctly into the PVS syntax by the style sheet. Therefore, the “standard” OMDoc documents have to be transformed to a specific OMDoc format so that they can be accepted by the interface. Of course, the style sheet for PVS, namely *omdoc2pvs.xsl*, does not support some data types such as array, enumeration either and also has to be modified to support those data types.

Thus, the translation actually involves three steps instead of the ideal two: the first step is to translate FDL into the OMDoc document containing both standard and private CDs; the second step is to manipulate and customize the document into the specific OMDoc document; the last step is to apply the modified PVS-based XSLT style sheet to transform the specific OMDoc document into the PVS language syntax. To distinguish these two kinds of OMDoc documents, in this thesis, we call the first OMDoc document as the *generic OMDoc document* and the specific OMDoc document as the *PVS-oriented OMDoc document*.

The following Figure 6.1 from a high-level perspective illustrates the whole translation process from SPARK FDL into PVS via the OMDoc approach.

Figure 6.1 A high-level diagram of translation from SPARK FDL into PVS.

The following sections will discuss the translation in detail, especially problems we encountered, the solutions for those problems, and an example of the final translation result. The last section will briefly discuss how the translation was implemented.

6.2 Translating FDL into the generic OMDoc

Recall from Chapter 2, SPARK FDL specifications are actually generated and allocated in three separate files. So a question might arise: what is relationship between the SPARK FDL specifications and those OMDoc elements?

As described in Chapter 5, the *symbol* element specifies a symbol of a mathematical concept, and the *definition* element gives a “fixed” meaning of such a symbol. Therefore, all FDL declarations in the *.fdl* file are translated into the *symbol* and *definition* element part (*var* declarations are exceptional, and the explanation will be discussed in the later section). The *axiom* element, unlike the *definition* element, which fixes the meaning of a symbol, only restricts the possible interpretations of the symbol. Therefore, in practice, these *axiom* elements can be used to specify FDL rules to constrain corresponding FDL declarations. Thus, all FDL rules in the *.rls* file become the *axiom* element parts. Finally, all FDL verification conditions in the *.vcg* file become the *assertion* element parts. The following figure demonstrates a high-level overview of translation process from SPARK FDL files to an OMDoc document.

6.2.1 Translating FDL constant declarations

Suppose given three FDL files, `swapelement.fdl`, `swapelement.rls` and `swapelement.vcg`, which are generated from `swapelement.ada` (see Figure 2.1) by the SPARK tool -- Examiner, are re-illustrated in the following Figures from Chapter 2:

```

title procedure swapelement;

function round__(real) : integer;
type atype = array [integer] of integer;
type index = integer;
const index__last : integer = pending;
const index__first : integer = pending;
const integer__last : integer = pending;
const integer__first : integer = pending;
var a : atype;
var j : integer;
var i : integer;

end;
```

Figure 6.2 An FDL Declarations file (`swapelement.fdl`).

```

rule_family swapelement_rules:

  X    requires [X:any] &
  X <= Y requires [X:ire, Y:ire] &
  X >= Y requires [X:ire, Y:ire].

swapelement_rules(1): character__pos(X) may_be_replaced_by X.
swapelement_rules(2): character__val(X) may_be_replaced_by X.
swapelement_rules(3): integer__first <= integer__last may_be_deduced.
swapelement_rules(4): integer__base__first <= integer__base__last may_be_deduced.
swapelement_rules(5): integer__base__first <= integer__first may_be_deduced.
swapelement_rules(6): integer__base__last >= integer__last may_be_deduced.
swapelement_rules(7): index__first may_be_replaced_by 1.
swapelement_rules(8): index__last may_be_replaced_by 10.
swapelement_rules(9): index__base__first <= index__base__last may_be_deduced.
swapelement_rules(10): index__base__first <= index__first may_be_deduced.
swapelement_rules(11): index__base__last >= index__last may_be_deduced.
```

Figure 6.3 An FDL rules file (`swapelement.rls`).

For path(s) from start to finish:

procedure_swapelement_1.

```

H1: true .
H2: i >= index__first .
H3: i <= index__last .
H4: j >= index__first .
H5: j <= index__last .
H6: for_all (i__1: index, ((i__1 >= index__first) and (
    i__1 <= index__last)) -> ((element(a, [i__1]) >=
    integer__first) and (element(a, [i__1]) <=
    integer__last))) .
->
C1: update(update(a, [i], element(a, [j])), [j], element(
    a, [i])) = update(update(a, [i], element(a, [j])), [
    j], element(a, [i])) .

```

Figure 6.4 An FDL Verification Conditions file (swapelement.vcg).

From the examples Figure 5.1 and 5.2 that illustrated in Chapter 5 we have seen, the FDL function declaration can be specified by using the *symbol* element, and an FDL type identifier and its specification can be specified by using the *symbol* and *definition* elements respectively. Therefore, the FDL reserved words such as ‘function’ and ‘type’ can be omitted during the translation since OMDoc uses <OMS> element’s two attributes *cd* and *name* to distinguish the different declarations. For the same reason, the two reserved words ‘const’ and ‘var’ are also deliberately omitted during the translation. However, some special issues are needed to be considered during the translation of FDL constant and variable declarations. The rest of this section mainly talks about how the FDL constant declaration is dealt with and leaves the discussion of variable declaration to the next section.

As discussed in Chapter 2, instead of explicitly assigned actual values, all SPARK FDL constants are declared as pending. The actual value of each constant is provided by the corresponding the FDL rules file. This is a complication because the actual SPARK FDL allows a constant declaration to be assigned a value directly. The generated FDL constant declarations, however, are always declared as pending. Therefore, in practice, the reserved

word ‘pending’ in an FDL constant declaration should be omitted as well because pending information can be decided later in the FDL rules file. As a result, the constant identifier and its type are still translated into the *symbol* element part and rules that explain the pending information in the *.rls* file are translated into the *axiom* element part.

Example:

An integer constant `index__first` declared in Figure 6.2

```
const index__first : integer = pending;
```

and its corresponding pending information can be obtained from the FDL *.rls* file in Figure 6.3: `swapelement_rules(7): index__first may_be_replaced_by 1.`

They can be expressed as the following OMDoc form:

```
<symbol id="index__first" kind="object" scope="global">
  <type system="sts">
    <OMOBJ>
      <OMS cd="setname1" name="Z"/>
    </OMOBJ>
  </type>
</symbol>
<axiom id="swapelement_rules(7)">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="relation1" name="eq" />
        <OMS cd="swapelement" name="index__first" />
        <OMI>1</OMI>
      </OMA>
    </OMOBJ>
  </FMP>
</axiom>
```

Figure 6.5 The translation result from an FDL constant into an OMDoc form.

As illustrated from Figure 6.5, OpenMath has a symbol, namely *Z* from a CD that called *setname1*, expressed by `<OMS cd="setname1" name="Z"/>` to represent the set of integers, positive, negative and zero. Therefore, an FDL integer type is translated as the OpenMath’s symbol *Z*. The reserved word *may_be_replaced_by* from the FDL rules file is translated as the equation symbol *eq* from *relation1* CD, `<OMS cd="relation1" name="eq" />`. More FDL

rules will be discussed in section 6.2.4. The constant identifier `index__first` is specified as a symbol in the swapelement theory, so it is translated as a symbol (`<OMS cd="swapelement" name="index__first" />`) as well. Otherwise, an identifier is translated into the `<OMV>` element.

It deserves to be pointed out that picking CDs is not trivial as well. OpenMath contains enormous symbols in various CDs, and some of them are very close or even have same mathematical meaning. Therefore, choosing CDs is a technical non-trivial and time-consuming job. For instance, OpenMath has two identical symbols *type* in two separate CD groups. One CD is called *mathmltypes*, which is compatible with the standard MathML (e.g. `<OMS cd="mathmltypes" name="type"/>`). Another is called *sts*, which represents for the OpenMath *Small Type System* (e.g. `<OMS cd="sts" name="type"/>`). Both of these two symbols can be used to specify the type of an object. Take another example, the *mathmltypes* CD also hold various types (e.g. `<OMS cd="mathmltypes" name="integer_type"/>` stands for the integer type). As discussed earlier, OpenMath, however, contains another CD group -- *setname1*, to define common sets of mathematics [22]. The symbol *Z* also can be used to represent an integer type. OpenMath itself has no recommendations or restrictions on choosing which symbols to represent integers. Thus, either *integer_type* or *Z* can be used as an attribute to convey the type of an integer.

6.2.2 Dealing with FDL variable declarations

Variables in FDL declarations do not need to be translated into *symbol* elements because they simply provide a name and associated type so that formulae can be succinct. For instance, given a variable declaration `var x: int` and an expression `H1:  $x+1 > x$` in a *.vcg* file. The expression simply implies (*for all x , $x+1 > x$*).

Due to this reason, FDL variable declarations can be translated as binding expressions using the OpenMath universal quantifier `for_all` whenever those variables occur in the corresponding `.vcg` file. Therefore, those variables can be annotated as OMDoc comments at the declaration section.

The detailed translating process is as follows:

First, all FDL variable declarations are translated into the OMDoc *omtext* elements. To distinguish with other OMDoc comments, the translation uses each FDL variable identifier with prefix `'var_'` as the value of *id* attribute of the *omtext* element. The type specification of each variable is translated into the `<CMP>` element section as the child of the *omtext* element. The prefix `'var_'` serves another purpose. These variables are only occurred in FDL expressions in the verification conditions (`.vcg`) file. So the binding information of each variable is only needed to be added in the *assertion* element section during the translation by looking up each variable declared in the `.fdl` file. This is done by making traversals through the *assertion* element searching for these variables. Once a new identical identifier is found, a binding expression is added ahead of such identifier expression in the OMDoc expression. The reason that only new identifiers are added is to avoid redundancy. It is possible that the same identifier might occur in each FDL theorem in the `.vcg` file more than once, but it is only required to be quantified once. Otherwise, the translation result for the binding information ends with incorrect expression.

Example:

Given variable declarations from Figure 6.2:

```
var a : atype;  
var j : integer;  
var i : integer;
```


At first, these three variables are translated into OMDoc form, but in here we only list the corresponding OMDoc fragment for the last variable i and its type integer:

```
<omtext id="var_i">
  <CMP>
    <OMOBJ>
      <OMS cd="setname1" name="Z"/>
      <OMV name="i" />
    </OMOBJ>
  </CMP>
</omtext>
```

Since the variable i is not specified as the *symbol* element, it is only translated as the <OMV> element.

For the first two expressions containing variables i

H2: $i \geq \text{index_first}$.

H3: $i \leq \text{index_last}$.

from Figure 6.4, they are translated as the following generic OMDoc fragment from line 32 to 41:

```
<assertion id="procedure_swapelement_1" type="theorem">
  <FMP>
    <OMOBJ>
      <OMBIND>
        <OMS cd="quant1" name="forall" />
line 6    <OMBVAR>
line 7    <OMATTR>
          <OMATP>
            <OMS cd="sts" name="type" />
            <OMS cd="setname1" name="Z"/>
          </OMATP>
line 12   <OMV name="i" />
line 13   </OMATTR>
          <OMATTR>
            <OMATP>
              <OMS cd="sts" name="type" />
              <OMS cd="setname1" name="Z"/>
            </OMATP>
            <OMV name="j" />
          </OMATTR>
          <OMATTR>
            <OMATP>
              <OMS cd="sts" name="type" />
              <OMS cd="swapelement" name="atype" />
            </OMATP>
            <OMV name="a" />
          </OMATTR>
line 28   </OMBVAR>
          <OMA>
            <OMS cd="logic1" name="and" />
            <OMS cd="logic1" name="true" />
line 32   </OMA>
```

```

line 34    <OMS cd="relation1" name="geq" />
           <OMV name="i" />
           <OMS cd="swapelement" name="index__first" />
           </OMA>
           <OMA>
line 39    <OMS cd="relation1" name="leq" />
           <OMV name="i" />
           <OMS cd="swapelement" name="index__last" />
line 41    </OMA>
           ...
           </OMA>
           </OMBIND>
           </OMOBJ>
           </FMP>
           </assertion>

```

Figure 6.6 The OMDoc fragment for FDL variable declarations and their binding information.

During the traversals, once the first identical variable identifier *i* is found (by comparing with the substring value of *id* attribute of the *omtext* element) at line 34, its corresponding binding information is added from line 7 to line 13. The traversal continues, and the second identical variable identifier *i* is found at line 39, but since it is not a new variable anymore, no binding information is added at this time. The traversal continues until it reaches the `</assertion>` element. In Figure 6.2, there are total three variables *a*, *j* and *i*, but their appearances in the VCs expression take the order as *i*, *j* and *a* respectively in Figure 6.4. Thus, the binding information of these three variables is added from line 6 to line 28 respectively in Figure 6.6.

Eventually, such binding information can be expressed in PVS syntax as:

FORALL(*i*: int, *j*: int, *a*: atype):.....

6.2.3 Translating FDL type declarations

As described in Chapter 2, SPARK FDL has two kinds of types: *simple types* and *structured types*. Simple types comprise the *boolean*, *integer*, *real* and *enumeration* type. Structured types include

the *array* and *record* type. However, Content Dictionaries in OpenMath and OMDoc do not have CDs to directly encode structured types and the simple *enumeration* type. Fortunately, both OpenMath and OMDoc allow the specification of “private” CDs. Therefore, we can define our “private” CD to support these types. For the sake of consistency, these data type declarations are grouped in the same CD, namely *fdl* in the *cd* attribute. As for the value of the *name* attribute, it can take *enumeration_type* or *record_type* as its value depending on different types. In the OMDoc, however, both the <OMV> and <OMSTR> elements can be used to specify the *components* of the *record type* but without any explanation. We believe this inconsistency is a flaw of OMDoc and it causes confusion. We argue that the *components* of each *enumeration* and *record type* should be translated as <OMSTR> elements rather than <OMV> elements.

It also should be pointed out that SPARK FDL has no restrictions that the identifiers of each component and field of a record type cannot be the identifiers of variables. In other words, the identifiers of components and variables can be identical.

Example:

The following FDL declaration segment contains both a record type and variables:

```
type stacktype = record
  vector : vectors;
  ptr : integer
end;
var state : stacktype;
var vector : vectors;
var ptr : integer;
```

the identifiers of the components of the record type (i.e. vector and ptr) are identical with those of variables in above example.

Those variables (such as state, vector and ptr) in above example are still translated as <OMV> (see section 6.2.2). But the components (vector and ptr) of the record type

stacktype are translated as <OMSTR> rather than <OMV>.

The record type stacktype declaration is translated as the following OMDoc segment:

```
<symbol id="stacktype" kind="type" scope="global">
  <type system="sts">
    <OMOBJ>
      <OMS cd="sts" name="type" />
    </OMOBJ>
  </type>
</symbol>
<definition id="stacktype" for="stacktype" type="simple">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="fdl" name="record_type" />
        <OMATTR>
          <OMATP>
            <OMS cd="sts" name="type" />
            <OMS cd="type" name="vectors" />
          </OMATP>
          <OMSTR>vector</OMSTR>
        </OMATTR>
        <OMATTR>
          <OMATP>
            <OMS cd="sts" name="type" />
            <OMS cd="setname1" name="Z" />
          </OMATP>
          <OMSTR>ptr</OMSTR>
        </OMATTR>
      </OMA>
    </OMOBJ>
  </FMP>
</definition>
```

Figure 6.7 The OMDoc fragment for the FDL record type declaration.

Also illustrated in Figure 6.7, <OMS cd="fdl" name="record_type" /> in the *definition* element is an example of user-defined “private” CD.

As for the SPARK FDL *array* type, one of the alternatives is, as with the *enumeration* type, to specify an array as a symbol, e.g. <OMS cd="fdl" name="array_type" />. But in this case, eventually, the corresponding style sheet has to be modified as well to proper transform the “private” array CD and symbol. Another alternative is to eliminate the reserved word *array*,

leave it as a standard OpenMath *function type* (since from a mathematical perspective, an array type can be considered as a function type). Since the target translation language is PVS, and PVS considers an array type as a function type as well. In addition, the style sheet does not need to be updated as well. Therefore, it is much easier for us to choose the latter case.

Example:

Given an FDL array type declaration in Figure 6.2

```
type atype = array [integer] of integer;
```

it can be expressed as the following OMDoc fragment (the symbol element section is omitted):

```
<definition id="atype" for="atype" type="simple">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="mathmltypes" name="fn_type" />
        <OMS cd="setname1" name="Z" />
        <OMS cd="setname1" name="Z" />
      </OMA>
    </OMOBJ>
  </FMP>
</definition>
```

Figure 6.8 The OMDoc fragment for the FDL array type declaration.

In Figure 6.8, the FDL array type is translated into the OMDoc function type expressed by the symbol `<OMS cd="mathmltypes" name="fn_type" />`.

6.2.4 Dealing with FDL rule declarations

As described in section 2.2.6 in Chapter 2, FDL rules consist of the rule family and rule declarations. Those declarations are generated by the SPARK tool – Examiner and are supplied to other tools, such as the SPARK Simplifier and the Proof Checker to conduct further verification. However, although the rule family provides type constraints for each rule

that contains variables (e.g. either X or Y in Figure 6.3), the specific type of each variable is dynamically determined by the SPARK tools when a rule is applied.

For instance, given the rules declaration from Figure 6.3 re-illustrated as follows:

```
rule_family swapelement_rules:
  X requires [X:any] &
  X <= Y requires [X:ire, Y:ire] &
  X >= Y requires [X:ire, Y:ire].

swapelement_rules(1): character__pos(X) may_be_replaced_by X.
swapelement_rules(2): character__val(X) may_be_replaced_by X.
swapelement_rules(3): integer__first <= integer__last may_be_deduced.
swapelement_rules(4): integer__base__first <= integer__base__last may_be_deduced.
swapelement_rules(5): integer__base__first <= integer__first may_be_deduced.
swapelement_rules(6): integer__base__last >= integer__last may_be_deduced.
swapelement_rules(7): index__first may_be_replaced_by 1.
swapelement_rules(8): index__last may_be_replaced_by 10.
swapelement_rules(9): index__base__first <= index__base__last may_be_deduced.
swapelement_rules(10): index__base__first <= index__first may_be_deduced.
swapelement_rules(11): index__base__last >= index__last may_be_deduced.
```

Figure 6.9 An example of the SPARK FDL rules declaration.

The actual types of variables that are declared in the beginning of `rule_family` section (such as X in the above example) in Figure 6.9, are dynamically further determined by the SPARK tools in the proof process. Actually, rule family declarations are not real declarations in SPARK FDL. They merely provide some type constraints. For instance, the rule family declaration

$X \geq Y$ requires $[X:ire, Y:ire]$

infers that if any rules containing two variables have the same pattern (i.e. $\geq$) with above declaration, whatever matches the two variables that can only be of *integer*, *real* or *enumeration* type. The worst case is the enumeration type. From the rule family declaration, we can only know two variables are both enumeration types, but we know nothing about the concrete contents of the enumeration type (i.e. the enumeration type can be (e_1) , (e_1, e_2) , (e_1, e_2, e_3) , ...). At this point, we cannot translate them properly into the OMDoc document.

In the succeeding rules declaration section, the first two rules contain `character__pos` and

character_val expressions. They are SPARK FDL built-in function attributes. As for the expressions containing the same suffixes such as __base__first and __base__first (i.e. in rules 4,5,6,9,10 and 11) are FDL built-in constant attributes. These attributes come from SPARK Ada attributes (e.g. SPARK Ada language supports attributes such as Pos, Val, Base [23]), so they are not even explicitly defined in SPARK FDL files. Although they can be translated into the OMDoc form, while transforming into other languages, their syntax will not be able to be recognized unless the specification information of these attributes is explicitly present.

Fortunately, experience has shown that the rules that contain both function and constant attributes are not invoked by the SPARK verification tools. In another words, these rules are not used during the SPARK verification process. Also, we assume that other theorem proving systems should generate their own rules or axioms based on their own rationale rather than the SPARK system. Therefore, the rule family dummy declarations such as:

```
X    requires [X:any] &
X <= Y requires [X:ire, Y:ire] &
X >= Y requires [X:ire, Y:ire].
```

are eliminated during the translation. Also, all of the rules that contain both function and constant attributes are eliminated as well. Unfortunately, this is a side effect of the translation.

As a result, only those rules that specify the meanings of ‘pending’ information in FDL constant declarations are retained. For instance, in Figure 6.3, the FDL swapelement rules file, only rule 3, 7 and 8 are translated into the OMDoc form.

As addressed earlier, SPARK FDL rules declarations are translated as the OMDoc *axiom* element section. Each FDL rule identifier becomes the value of *id* attribute of the *axiom* element, the content of each rule is translated into the <FMP> element section. Reserved

words such as ‘may_be_replaced_by’ and ‘are_interchangeable’ are translated as the OpenMath *eq* (meaning binary equality) symbol in the *relation* CD and the *equivalent* symbol in the *logic* CD respectively.

Example:

swapelement_rules(7): index__first may_be_replaced_by 1.

is translated as the following OMDoc expression:

```
<axiom id="swapelement_rules(7)">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="relation1" name="eq" />
        <OMS cd="swapelement" name="index__first" />
        <OMI>1</OMI>
      </OMA>
    </OMOBJ>
  </FMP>
</axiom>
```

The reserved word ‘may_be_deduced_from’ is translated as the *implies* symbol in the *logic* CD in reversed order. For instance, an FDL expression ‘A may_be_deduced_from B’ is translated as the following OMDoc expression:

```
<OMS cd="logic" name="implies" />
<OMV " name="B" />
<OMV " name="A" />
```

As for another reserved word, ‘may_be_deduced’ is simply omitted.

Aside from SPARK generated rules, it should be pointed out that SPARK also allows users to define their own proof rules if necessary. These additional proof rules may include operators and use-defined proof functions. For instance, SPARK FDL has ‘mod’ operator. However, unlike other arithmetic operators that are supported by the various SPARK built-in arithmetic rules, the ‘mod’ operator has no any built-in rules provided by SPARK [31]. Users have to provide additional proof rules for the ‘mod’ operator when necessary to conduct the verification. The additional proof rules have the same structure and format with built-in

rules as well as those generated proof rules. However, to distinguish from those generated rules, the user self-defined rules are saved in a file with extension *.rul* rather than *.rls*. Similar to the problems as discussed earlier, the concrete variables and data types are not always explicitly specified in the user-defined rules either. For instance, a user-defined rule may include a proof function $FOO(x, y)$ (e.g. $X \leq Y$ requires $[X:ire, Y:ire]$, $FOO(x, y)$ may_be_replaced_by $X \leq Y$). Similar to those generated rules, the variable x and y of $FOO(x, y)$ can take various type (i.e. *integer*, *real*, *enumeration* or any other types). The rule about the proof function $FOO(x, y)$ could be decomposed as

```
forall (x:integer, y:integer), FOO(x, y) <=> x<=y
forall (x:real, y:real), FOO(x, y) <=> x<=y
forall (x:e1, y: e1) FOO(x, y) <=> x<=y
forall (x:e2, y: e2) FOO(x, y) <=> x<=y
forall (x:e3, y: e3) FOO(x, y) <=> x<=y
...
```

where $e1, e2, e3 \dots$ are all the enumeration types defined in the FDL file. These formulas can then be translated into the corresponding OMDoc axioms. This part of the translation was not implemented since the intended use of the translator does not involve user-defined proof-function rules.

6.2.5 Translating FDL Verification Conditions

In the FDL verification conditions file, the optional traceability comments (e.g. ‘For path(s) from start to finish:’) in Figure 6.4 can either be simply omitted or translated as typical OMDoc comments. A *procedure* or *function path identifier* such as `procedure_swapelement_1` in Figure 6.4 is translated as the value of *id* attribute of the *assertion* element. Since each VC in the *.vcg* file is also a theorem, the *type* attribute of the *assertion* element takes “theorem” as its

value: `<assertion id="procedure_swapelement_1" type="theorem">`. Each header of hypotheses and conclusions that are denoted by ‘Hi’ and ‘Ci’ respectively is also omitted and only the contents of every hypothesis and conclusion are translated into the OMDoc document.

The contents of each VC or theorem in the FDL *.vcg* file contain various FDL expressions that were described in Chapter 2. Many FDL expressions can be easily translated into OMDoc because the standard OpenMath has CDs to support symbols, such as arithmetic, logic operators as well as quantifiers. But there are also some expressions that cannot be directly translated into OMDoc. For instance, as described in Chapter 2, FDL uses `fld_` and `upf_` as prefixes to access the components or fields of a record type. Another prefix `mk_` is used to assemble objects of a structured type in FDL. Since these prefixes are used only in SPARK FDL to access the components or fields of structured types, they are not even specified in the FDL declarations file. Therefore, even if they could be translated into OMDoc, other proof systems (including PVS) still will not be able to handle them. Besides, since OMDoc allows the definition of “private” CDs, similar to our `record_type` symbol (see 6.2.3) created in the FDL declarations, we can create a symbol for these structured type expressions. For instance, to distinguish the symbol of record type, we use a symbol, namely *recordaccessors* in the *fdl* CD (`<OMS cd="fdl" name="recordaccessors" />`) to denote a record expression (not record type). So even without these prefixes, OMDoc still can handle them properly. Furthermore, those private symbols can be rendered into appropriate expressions later by applying the corresponding XSLT style sheets. Thus, these FDL prefixes can be eliminated during the translation.

Similar to the structured types declaration in the *.fdl* file, the components or fields of structured types are translated as `<OMSTR>` instead of `<OMV>` as well.

Example:

Given an FDL expression as follows:

H2: `vector = fld_vector(state) .`

The corresponding OMDoc expression will be:

```
<OMA>
  <OMS cd="relation1" name="eq" />
  <OMV name="vector" />
  <OMA>
    <OMS cd="fdl" name="recordaccessors" />
    <OMV name="state" />
    <OMSTR>vector</OMSTR>
  </OMA>
</OMA>
```

In above example, `<OMS cd="fdl" name="recordaccessors" />` is another user defined private CD for accessing record expression. As for the expression `fld_vector`, it is translated as `<OMSTR>vector</OMSTR>`.

FDL expressions for manipulating the array type such as `element` and `update` are translated as corresponding private symbols.

Example:

Given an FDL VC expression as follows:

`update(a, [i], element(a, [j]))`

the corresponding OMDoc expression will be:

```
<OMA>
  <OMS cd="fdl" name="update" />
  <OMV name="a" />
  <OMV name="i" />
  <OMA>
    <OMS cd="fdl" name="element" />
    <OMV name="a" />
    <OMV name="j" />
  </OMA>
</OMA>
```

As a summary, the translation result from the example of three FDL swapelement files into the generic OMDoc format can be found in Appendix C.

6.3 Translating the Generic OMDoc to the PVS-oriented OMDoc

As mentioned earlier, OMDoc has alleged it contained an interface that can transform OMDoc documents into the PVS language syntax. The interface includes both an XSLT style sheet, namely *omdoc2pvs.xsl* and the PVS *prelude library*. The prelude library contains many fundamental mathematical theories (such as booleans, integers, reals etc) in the OMDoc format so that users do not need to redefine these theories. Any symbols declared in the prelude library can also be directly referenced in user defined OMDoc documents. However, instead of using the standard OpenMath CDs, these symbols are declared by using variant CDs. For instance, the standard OpenMath uses `<OMS cd="setname1" name="Z"/>` to express an integer, while the prelude library uses `<OMS cd=" integer" name="int "/>`. Therefore, the translated generic OMDoc documents containing the standard OpenMath CDs are not even accepted by the interface. This means that generic OMDoc documents cannot directly be transformed into the PVS language syntax. They have to be transformed into a format that can be recognized by the interface. To distinguish with the generic OMDoc document, we call such format as the *PVS-oriented OMDoc* document in this thesis.

The OMDoc project itself did not mention anything about why they chose variant CDs instead of using the standard OpenMath CDs. We assume that since PVS uses such as `int` and `bool` to denote an integer and boolean, it is much easier for the prelude library in the form of OMDoc using such variant CDs rather than the standard OpenMath CDs. However, the PVS-oriented OMDoc significant weakens the OMDoc as a so-called universal language or an interlingua.

One of the alternatives is to rewrite the whole prelude library and the PVS-based style sheet using the standard OpenMath CDs format. But the PVS prelude library provided by OMDoc contains almost one hundred theorems. So this may leave to the future work.

6.3.1 Changing attribute values of <OMS> elements

The most significant difference between the generic and the PVS-oriented OMDocs is attribute values. All of the symbols that are referenced from the standard OpenMath CDs have to be transformed into the interface accepted format. The reason is because they will be transformed by the interface with the PVS prelude library supported built-in types or reserved words, such as 'int', 'bool' 'AND' and 'OR' etc. The complete list of PVS-oriented OMDoc CDs are omitted due to the space reason, they can be referenced from [36].

The OpenMath standard *Small Type System* symbol (<type system="sts">) should be changed to <type system="pvs">. The private CD 'fdl' is also changed to 'pvs' as well.

As for the symbols that reference the *theory* element (e.g. <OMS cd="swapelement" name="index__first" />), there is no need to change them because they are dynamically generated by corresponding theories and it will be transformed by the interface as the typical PVS identifiers, or regular expressions.

6.3.2 FDL mod operator

FDL has 'mod' operator. It takes two integers as operands, and the result is also an integer. However, although PVS has 'rem' operator similar to the 'mod' operator in FDL, it only

takes positive integers as operands. Therefore, the FDL ‘mod’ operator cannot be directly translated into PVS ‘rem’ operator. A further restriction must be given to the ‘mod’ operator. Fortunately, PVS supports ‘abs’ (absolute operation) as well, so for each theory, the FDL ‘mod’ operator can be restricted as the equivalent PVS ‘rem’ operator by using the ‘abs’ operation. The snippet of FDL ‘mod’ operator in the OMDoc format is omitted, please refer to Appendix D.

Finally, an example of the swapelement specifications in the form of PVS-oriented OMDoc is illustrated in Appendix D.

6.4 Transforming from PVS-Oriented OMDoc into PVS

As illustrated earlier, the interface provided by OMDoc can transform the PVS-oriented OMDoc into the PVS language syntax. Acting as an interpreter, the *omdoc2pvs.xsl* style sheet is responsible for parsing OMDoc documents. During the parsing, for each symbol declared in the <OMS> element, the style sheet compares it with those in the prelude library. If they are identical (including both values of *cd* and *name* attributes), they will be transformed into the corresponding PVS prelude library supported types (such as int, bool) or the PVS reserved words, otherwise they will be transformed as regular identifiers or expressions.

However, OMDoc is still in the development phase. The *omdoc2pvs.xsl* XSLT style sheet does not have the presentation features of PVS, such as array and enumeration types. Thus, as a supplement, this experimental translation also has implemented a modified version of *omdoc2pvs.xsl* XSLT style sheet to support these types. The detailed discussion of the modified version style sheet is omitted because it involves extensive programming of XSLT

language and it not the main focus of this thesis.

By applying the modified PVS oriented XSLT style sheet, the PVS-oriented OMDoc in Appendix D can be transformed into the following PVS syntax:

```
swapelement: THEORY
BEGIN

%% % define fdl mod function using PVS rem funnction
mod: [int,int -> int] = (lambda(x: int, y: int): rem(abs(x),abs(y)));
%% % procedure swapelement
round_: [real -> int];
atype: TYPE = [int -> int];
index: TYPE = int;
index__last: int;
index__first: int;
integer__last: int;
integer__first: int;
%% %
%% % rule_family swapelement_rules:
swapelement_rules(3): AXIOM <=(integer__first,integer__last);
swapelement_rules(7): AXIOM =(index__first,1);
swapelement_rules(8): AXIOM =(index__last,10);
procedure swapelement_1: THEOREM (forall (i: int, j: int, a: atype): ) AND (TRUE, >=(i,
index__first),<=(i,index__last),>=(j,index__first),<=(j,index__last),(forall(i__1: index):
IMPLIES (AND(>=(i__1, index__first), <=(i__1,index__last)), AND (>=(a(i__1),
integer__first), <=(a(i__1), integer__last))))IMPLIES(AND(=(((a WITH [ (i) := a(j)]) WITH
[ (j) := a(i)]), ((a WITH [ (i) := a(j)]) WITH [ (j) := a(i)]))));
END swapelement
```

Figure 6.10 The PVS language representation for swapelement theory.

As illustrated in Figure 6.10, the generated PVS syntax by the *omdoc2pvs.xsl* XSLT style sheet for PVS uses prefix to represent an expression instead of infix fixity, so an expression $i \leq \text{index_last}$ becomes $\leq(i, \text{index_last})$. Fortunately, PVS supports both prefix and infix expressions and can handle the generated prefix expression.

The PVS language syntax has variable declarations, but because the SPARK FDL variables are translated as a PVS binding expression, instead of explicitly specified in the declaration part, these variables are specified as a binding expression in the THEOREM section.

Furthermore, it should be pointed out that the outputs generated from the style sheet look ugly. Although the outputs can be recognized by the PVS system as valid syntax, they are difficult to understand for human readers since the theorem prover in the PVS system is an interactive verification tool and needs user inputs to conduct the verification. As illustrated in Figure 6.10, even for a simple program like `swapelement`, the generated PVS language syntax is too obscure for human readers to understand.

6.5 Implementation

For the sake of completeness, this section discusses the translation from the implementation perspective.

6.5.1 Tools, Language and XML Processing Approach

Since an OMDoc document is a XML application, the Java programming language becomes the unbeatable choice because Java has a native mechanism for processing XML applications. This mechanism is called the Java API for XML Processing (JAXP) [38]. JAXP has built-in functionalities to parse and navigate XML applications depending on different purposes.

Generally, there are two major standard APIs for processing XML documents with Java -- the Simple API for XML (SAX) and the Document Object Model (DOM) [38]. In addition there are a host of other, somewhat idiosyncratic APIs including JDOM, dom4j, ElectricXML, and XMLPULL as well [39]. SAX uses event-driven, serial-access mechanism

for accessing XML documents, and it usually requires accessing the entire document at once in order to take action. Unlike SAX, DOM is a read-write API based on a tree-structure approach; one can create nodes, remove nodes, change their contents, and traverse the node hierarchy. Since we are going to generate the generic OMDoc document first and then will manipulate and customize it into the specific PVS-oriented OMDoc document, the DOM approach seems to be our perfect selection. However, because DOM is a cross-language API and is designed for reconciling differences between the object models implemented by different vendors, this makes it less than ideal as a Java API for processing XML [39]. Fortunately, similar to DOM, JDOM [40], is also a tree-based Java API for processing XML documents. It is designed purely for Java and XML, with no concern for backwards compatibility with earlier APIs. For instance, the following examples illustrate how to create a XML element by using JDOM and DOM respectively:

JDOM Example:

```
Element element = new Element("root");
```

DOM Example:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  
DocumentBuilder builder = factory.newDocumentBuilder();  
DOMImplementation impl = builder.getDOMImplementation();  
Document doc = impl.createDocument( null, "root_Numbers", null);  
Element element = doc.createElement("root");
```

From the above examples we can see, JDOM is much simpler than DOM. Also, unlike DOM, which has read-only restrictions on certain sections, a JDOM tree is fully readable and rewritten. All parts of the tree can be moved, deleted, and added to, subject to the usual restrictions of XML [39]. Thus, JDOM becomes an ideal approach to read, write and manipulate our OMDoc documents. Moreover, JDOM has been accepted as Java Specification Requests (JSR), which is defined in the formal Java specification [44]. For

detailed information on how to implement JDOM, please refer to the Chapter 14 of [39] and JDOM website at <http://www.jdom.org>.

Another issue is how to translate the source SPARK FDL language into the generic OMDoc document. The translation is done by applying classical compiler methodology [45]. More specifically, it uses a lexical analysis tool JFlex [41] and a syntactic parser generator tool CUP [42] to parse the lexical and syntactic meaning of SPARK FDL language as well as to generate the generic OMDoc document. JFlex, is a Java based lexical analyzer generator or scanner. The function of JFlex is to read the input stream as characters and based on the user-defined rules to produce a sequence of tokens (such as keywords, numbers, identifiers and special symbols) that are ready for a syntax parser to do the syntax analysis. CUP, a Java based Constructor of Useful Parsers (CUP for short), is a syntax parser generator and it serves the same role as the widely used parser generator YACC [43]. CUP obtains stream of tokens from the lexical analyzer, such as JFlex, and conducts the parsing analysis. The parsing analysis may include a number of tasks, such as collecting information about various tokens to put in a symbol table, analyzing semantic and syntactic meaning based on user-defined grammar rules and generating intermediate code (or technically called parser) as output. Unlike YACC, which is a C language based parser generator, the specifications and grammar rules of CUP can be embedded with Java code (and JDOM code as well), and CUP produces parsers that are implemented in Java as well. Moreover, both JFlex and CUP have an interface with each other, which makes it easier to incorporate code between them [41,42].

6.5.2 Implementation details

Parsing the FDL language involves creating lexical analyzers (or scanners) and syntax analyzers (or parsers) based on the FDL language specifications. As we have already seen in Chapter 2, there are three different kinds of FDL input (.fdl, .rls and .vcg) files, and each FDL declarations, rules and verification conditions file contains different lexical and

semantic specifications. Therefore, we need to create a lexical analyzer and syntax analyzer for each input FDL file respectively. During the parsing, the Java code that carries out the actual translations from FDL into the generic OMDoc is embedded at various points of the grammar rules in each CUP parser generator. Once the scanners and the parsers are constructed by JFlex and CUP, they can be used to parse the source FDL files, and to generate the generic OMDoc document as output. Then, various manipulations and customizations can be operated on the generic OMDoc document in order to generate the PVS-oriented OMDoc document.

The following figures illustrate the relations of each scanner and parser generator and their generated code.

Figure 6.11 Parsing FDL Declarations (.fdl file).

Figure 6.11 can be used as an example to explain the functionalities and relationships of

each scanner and parser. As for the FDL rule file and verification conditions file, the explanations of each program functionalities are ignored because parsing them produces the same structure with similar functionalities (See Figure 6.12 and Figure 6.13 respectively).

Figure 6.12 Parsing FDL rule file (.rls file).

Figure 6.13 Parsing FDL Verification Conditions file (.vcg file).

The functionalities and relationships of each program for parsing FDL declarations *.fdl* file are as follows:

fdl.flex: the scanner specifications or the scanner generator, to generate the FDL scanner – *FDLscan.java*. *fdl.flex* defines all the specifications that the scanner requires. It includes a series of macro declarations and lexical rules. Macros are abbreviations for regular expressions, used to make lexical rules easier to read and understand. Macros in the *fdl.flex* include white space, line terminator, identifier and comments etc. The lexical rules contain a series of rules based on the lexical rules of the FDL *.fdl* file. For instance, the rules specify each of token, like delimiter, keyword, separately.

FDLscan.java: the FDL scanner for *.fdl* declaration file, is generated by the *fdl.flex*, JFlex generator file.

`fdl.cup`: the parser generator, to generate the FDL parser – `FDLparser.java` and symbol table – `FDLsym.java`. `fdl.cup` provides all the specifications that the parser requires. It contains all terminal and non-terminal declarations, precedence and associativity declarations as well as the FDL declarations grammar rules based on the specifications of FDL language. It also contains customized java code to set up the scanner and parsing method as well as to provide a debugging and exception handling mechanism. Most importantly, the grammar section embeds java code that constructs the corresponding OMDoc elements at the various points of grammar rules to customize the parser.

`FDLparser.java`: the FDL parser, is generated by the `fdl.cup`, CUP generator file.

`FDLsym.java`: the symbol table, is generated by the `fdl.cup`, CUP generator file by calling `fdl.flex`. It contains a series of symbol constant code declarations, one for each terminal symbol.

`OmdocExpr.java`: includes a series of methods conforming with the OMDoc format expressions. It is called by the embedded java code at various points in the grammar rule section of `fdl.cup`.

Once the scanners and the parsers have been generated, they can be called by the main routine, `SparkPvs.java`. `SparkPvs.java` first reads the input source FDL files, and calls the three scanners and parsers to do the lexical and syntactical analysis for each FDL input file. It also generates the generic OMDoc elements at various points of each grammar rule during the parsing phase. After the parsing is finished, the three parsers have returned three OMDoc files in the memory. Since OMDoc can only have one root element `<omdoc>`, `SparkPvs.java` first deletes two OMDoc root elements and merges them with the third file

and then serializes those elements into the generic OMDoc as one of the output files. SparkPvs.java also performs various manipulations to the generic OMDoc and generates the PVS-oriented OMDoc document.

Figure 6.14 Parsing FDL Language and generating OMDoc files.

Finally, the modified XSLT Style sheet, *omdoc2pvs.xsl* transforms the PVS-oriented OMDoc into the PVS language. Figure 6.14 illustrates the whole translation process from source SPARK FDL into OMDoc (including the both generic and PVS-oriented OMDoc) and then into the PVS language.

Chapter 7 Conclusions

This final chapter discusses the summaries and limitations of this thesis, and describes future work.

7.1 Summaries and limitations

Our translation solution demonstrates that the integration between the SPARK and the PVS systems can be achieved via the OMDoc approach. The final translation results, although in inelegant format, are in valid PVS syntax. More importantly, as an intermediate result of the translation, the generated OMDoc documents provide a consistent XML tree structure that can be transformed into other formats by applying different XSLT style sheets. As a consequence, it is predictable that theoretically, the interoperability among different verification systems can be achieved via the OMDoc methodology. In addition, from the software engineering perspective, the mathematical theories in the form of OMDoc format provide a consistent structure and make them possible to be modularized and reused among different theorem proving systems.

On the other hand, our translation solution also illustrates some unexpected difficulties during the translation. These difficulties are caused by the two verification languages: SPARK FDL and PVS as well as the limitations and shortcomings of the OMDoc approach.

From the SPARK FDL language perspective, as introduced earlier, SPARK FDL is based on relatively simple *First-order Logic* and PVS is based on much richer *Higher-order logic*. Thus we originally thought the translation from SPARK to PVS would be easy and straightforward, but it is not the case. SPARK FDL is automatically generated from the SPARK language. Therefore, it inherits many built-in functions and attributes from the SPARK language as illustrated in the previous chapters. Also, as generated code, SPARK FDL carries ‘extra’ information such as ‘pending’ information for constant declarations, rule-family declarations etc. The extra information has to be either customized or eliminated during the translation. The elimination may cause side effects as discussed in the previous chapter. Although based on the simple *First-order Logic*, SPARK FDL as a verification language not only contains basic types such as *booleans*, *integers* and *reals* etc., but also includes data types, such as *enumeration* and *structured types* (i.e. *array*, *record*) as well as quantifiers. Some of them (i.e. *record*) have their unique syntactic expressions.

From the semantic and syntactical aspect, mapping the operators and expressions from FDL to PVS is not always equivalent either. Some extra customizations are need as we have seen the FDL *mod* operation and the expression for accessing the FDL *array type*.

From the OMDoc approach aspect, OpenMath and its extension OMDoc are mainly dealt with mathematical objects, symbols and expressions rather than specification and verification languages, so they do not have Content Dictionaries to encode certain kinds of data types, such as arrays, enumerations and records. These kinds of data types, however, are fundamental in SPARK FDL, PVS as well as other specification and verification languages. Therefore, as a supplement, the translation solution has to add some “private” Content Dictionaries to support encoding for such kinds of date types. Besides, the expressions for the components of enumeration and record type in OMDoc are inconsistent as well.

More surprisingly, the translated generic OMDoc document even cannot be transformed correctly into the PVS syntax by the OMDoc interface because OMDoc employed variant CDs in the interface rather than the standard CDs specified by OpenMath. Therefore, in order to be properly transformed into the PVS syntax, the generated generic OMDoc document has to be adopted as the PVS-oriented OMDoc document. As a consequence, the actual translation is taken three steps instead of the expected two steps. This extra cost dramatically weakens the OMDoc document as its so-called a universal language or an interlingua. Besides, there is an inconsistency between two elements `<OMSTR>` and `<OMV>` in OMDoc while specifying the components of enumeration and record types.

Furthermore, since we have to provide our “private” CDs and symbols to support those data types that are described earlier in order to proper transform the generated OMDoc document into the correct PVS syntax, we also have to modify the corresponding PVS style sheet that is originally provided by OMDoc in order to support the correct transformation for those data types.

Another deficiency about OMDoc is, as indicated in section 6.4, although it is machine-readable and understandable, the final generated PVS syntax is difficult to understand for human users. After all, the theorem proving tools such as PVS, requires user inputs to conduct the verification and proof.

Last, from the translated OMDoc we have seen, OMDoc has essentially tree structure. Even for a moderate specification program, an OMDoc document can grow to be very large, so that more space-efficient representations are needed. Currently, OMDoc is still in the developing phase, the OMDoc tree space-efficiency problem will be considered in the later OMDoc version [14].

7.2 Future work

The future work can lead to several directions.

First, as mentioned in the motivation section of Chapter 1, some real-time issues that cannot be dealt by the SPARK system can be conducted in the PVS system for the further reasoning and verification via the integration. Also, the PVS theorem prover can be used to verify and validate the correctness of translation.

More importantly, the PVS prelude library in the form of OMDoc can be rewritten into the standard OpenMath's format. As a result, the PVS style sheet provided by OMDoc is also needed to be updated so that the translation can be achieved via ideal two steps instead of three steps.

In addition, future work may involve to the further transformations from the translated generic OMDoc document into other verification systems, such as model checking systems, by implementing the appropriate XSLT style sheets.

On the other hand, OMDoc is a very large project. It contains rich symbols, statements and theory mechanisms. Therefore, many aspects of OMDoc mathematical statements, such as *proof representations*, *inheritance*, *complex theories* and other auxiliary elements are not covered in this thesis. For instance, in practice, especially for verifying large complex programs, not all definitions and axioms need to be explicitly stated in one theory; they can be imported or inherited from other theories. OMDoc supports theory inheritance mechanism, and it allows to importing inheritance information from other theories. This feature may be added by the further implementation if necessary.

Bibliography

- [1]. Sorensen, Reed, *"MIL-STD-498, and the CMM: How do they relate?"*, CrossTalk, November/December, 1995
- [2]. Jonathan Bowen, Victoria Stavridou, *"Safety-Critical Systems, Formal Methods and Standards"*, Software Engineering Journal, 8(4):189-209, July 1993.
- [3]. Ellsberger, J.; Hogrefe, D.; Sarma, A., *"SDL - Formal Object-oriented Language for Communicating Systems"*, Prentice Hall Europe, 1997, ISBN 0-13-621384-7.
- [4]. N.Platt, P.G.Larsen. *"An Overview of the ISO/VDM-SL Standard"*, Published by SIGPLAN Notices, Volume 27, Number 8, August 1992
- [5]. Mike Spivey. *"The Z Notation: a reference manual"*, Prentice Hall, International Series in Computer Science, 1992.
- [6]. The SMV System, available at <http://www-2.cs.cmu.edu/~modelcheck/smv.html>
- [7]. Gerard J. Holzmann, *"The SPIN Model Checker"*, Pearson Educational, September 4, 2003
- [8]. The PVS Specification and Verification System, available at <http://pvs.csl.sri.com>
- [9]. SPARKAda, Praxis Critical Systems Limited. <http://www.praxis-cs.co.uk/sparkada>
- [10]. Gordon, M.J.C, *"A Proof generating system for Higher-Order Logic"*, 1988
- [11]. Patrick Ion, David Carlisle, Robert Miner, and Nico Poppelier. *"Mathematical Markup Language (MathML) 2.0"*. W3C recommendation, 2001.
<http://www.w3.org/TR/MathML2>
- [12]. <http://www.w3.org>
- [13]. <http://www.w3.org/xml>
- [14]. Michael Kohlhase, OMDoc, *"An Open Markup Format for Mathematical Documents (Version 1.1)"*, June 5, 2003, available at <http://www.mathweb.org/omdoc/omdoc.ps>

-
- [15]. International Standards Organization, “*Reference Manual for the Ada Programming Language*”, ISO/IEC 8652: 1995
- [16]. Alan Burns, “*The Ravenscar Profile*”, ACM Ada Letters, 1999
- [17]. Peter Amey, “*High Integrity Ravenscar*” 2003. From Proceedings of Reliable Software Technologies – Ada Europe 2003. Lecture Notes in Computer Science Vol. 2655 Springer-Verlag 2003.
- [18]. Douglas J. Howe, and Stephen Michell, “*An approach to formal verification of real-time concurrent Ada programs*”, ACM Ada letters, 2003
- [19]. <http://www.w3.org/Style/XSL/>
- [20]. <http://www.praxis-cs.co.uk>
- [21]. John Barnes, “*High Integrity Ada, The SPARK Approach*”, Addison Wesley, 1997
- [22]. <http://www.openmath.org/cocoon/openmath//overview/index.html>
- [23]. PRAXIS critical Systems, “*SPARK95 – The SPADE Ada 95 Kernel*”, Edition 4.0, May, 2003.
- [24]. Michael Huth, Mark Ryan, “*Logic in Computer Science: Modelling and Reasoning about Systems*”, Cambridge University Press, 2000
- [25]. J. van Benthem and K. Doets, “*Higher-order logic, in Handbook of Logic for Computer Science I*”, edited by D. Gabbay and F. Guenther, Reidel, 1983
- [26]. SRI International, <http://www.sri.com/index.html>
- [27]. S. Owre, N. Shankar, J.M. Rushby and D.W.J. Stringer-Calvert, “*PVS Language Reference Version 2.4*”, SRI International, 2001
- [28]. <http://www.methweb.org/>
- [29]. Edmund M. Clarke and Jeannette M. Wing, “*Formal Methods: State of the art and future directions*”, ACM Computing surveys Vol. 28 No., 1996
- [30]. SPARK team, “*SPARK Examiner with Run-time Checker Generation of VCs for SPARK Programs*”, Issue 8.1, June, 2003
- [31]. SPARK team, “*SPADE Proof Checker, User Manual*” Issue 3.2, June, 2003

- [32]. Michael Kohlhase, “OMDoc, *An Infrastructure for OpenMath Content Dictionary Information*”, Bulletin of the ACM Special Interest Group on Symbolic and Automated Mathematics (SIGSAM), 34(2):43–48, 2000.
- [33]. <http://www.openmath.org/cocoon/openmath//overview/om-mml.html>
- [34]. O.Caprotti, D.P. Carlisle and A.M. Cohen, “*The OpenMath Standard*”, The OpenMath Esprit Consortium, February, 2000.
- [35]. Nico Poppelier, Robert Miner, Patrick Ion, David Carlisle, Ron Ausbrooks, Stephen Buswell, Stephane Dalmas, Stan Devitt, Angel Diaz, Roger Hunter, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. “*Mathematical Markup Language (MathML) 2.0 Specification*.” W3C Working Draft 20000328, April 1998. <http://www.w3.org/TR/REC-MathML2/>
- [36]. <http://www.mathweb.org/omdoc/index.html>
- [37]. <http://www.mathweb.org/mbase/index.html>
- [38]. Eric Armstrong, “*The Java API for Xml Processing (JAXP) Tutorial*”, Version 1.1, 2001, available at <http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/index.html>
- [39]. Elliotte Rusty Harold, “*Processing XML with Java*”, Addison-Wesley, 2002, available at: <http://www.cafeconleche.org/books/xmljava/>
- [40]. Brett McLaughlin and Jason Hunter, “*PVS Language Reference Version 2.4*”, SRI International, 2001
- [41]. Gerwin Klein, “*The Fast Lexical analyser Generator for Java*”, Version 1.3.5, 2001, available at: <http://www.jflex.de/>
- [42]. S.E. Hudson, “*CUP LALR Parser Generator for Java*,” (v0.10k), Sep. 1999, available at: <http://www.cs.princeton.edu/~appel/modern/java/CUP/>
- [43]. S. C. Johnson, “*YACC — Yet Another Compiler Compiler*”, CS Technical Report #32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
- [44]. Java Community Process, available at: <http://www.jcp.org/en/jsr/all>

- [45]. A. Aho, R. Sethi, and J. Ullman, "*Compilers: Principles, Techniques, and Tools*", Addison-Wesley Publishing, Reading, MA, 1986
- [46]. Christian Attiogbé, "*Formal Methods Integration for Software Development: Some Locks and Outlines*", Research reports from the Institut de Recherche en Informatique de Nantes are available at: <http://www.sciences.univ-nantes.fr/irin/Vie/RR/indexGB.html>, 2000
- [47]. Dr. Roderick Chapman, "*Industrial Experience with SPARK*", Praxis Critical Systems Limited, ACM SigAda 2000 conference.
- [48]. Hooker S., Lockyer M. A., and Fencott, C. (1996), "*CASE Support for Methods Integration: Implementation of a Translation from a Structured to a Formal Notation*", in *Methods Integration*, Leeds, 1996
- [49]. Payton, J., Gamble, R., Kimson, S., Davis, L., "*The Opportunity for Formal Models of Integration*". 2nd Int'l Conference on Information Reuse and Integration, 2000.
- [50]. Dan Zhou, Paul E. Black, "*Translating HOL TO Specifications for the Modle Checker SMV*", Edinburgh, Scotland, September 2001
- [51]. Andrew Adams, et "*Computer Algebra Meets Automated Theorem Proving: Integrating Maple and PVS*", Lecture Notes in Computer Science 2152, Springer, 2001
- [52]. Pavel Naumov, Mark-Oliver Stehr, Jos'e Meseguer, "*The HOL/NuPRL Proof Translator*", Lecture Notes in Computer Science 2152, Springer, 2001
- [53]. Pavel Naumov, "*Importing Isabelle Formal Mathematics into NuPRL*", TPHOLs'99: 12th International Conference, 1999
- [54]. Joe Hurd, "*Integrating Gandalf and HOL*", Lecture Notes in Computer Science 1690, Springer, 1999

Appendices

A. FDL reserved Words

are_interchangeable	as	assume
const	div	element
finish	first	for_all
for_some	goal	last
may_be_deduced	may_be_deduced_from	may_be_replaced_by
nonfirst	nonlast	not_in
odd	pending	pred
proof	requires	save
sequence	set	sqr
start	strict_subset_of	subset_of
succ	update	var
where		

B. PVS reserved Words

AND	CONJECTURE	FACT	LET	TABLE
ANDTHEN	CONTAINING	FALSE	LIBRARY	THEN
ARRAY	CONVERSION	FORALL	MACRO	THEOREM
ASSUMING	CONVERSION+	FORMULA	MEASURE	THEORY
ASSUMPTION	CONVERSION-	FROM	NONEMPTY_TYPE	TRUE
AUTO_REWRITE	COROLLARY	FUNCTION	NOT	TYPE
AUTO_REWRITE+	DATATYPE	HAS_TYPE	0	TYPE+
AUTO_REWRITE-	ELSE	IF	OBLIGATION	VAR
AXIOM	ELSIF	IFF	OF	WHEN
BEGIN	END	IMPLIES	OR	WHERE
BUT	ENDASSUMING	IMPORTING	ORELSE	WITH
BY	ENDCASES	IN	POSTULATE	XOR
CASES	ENDCOND	INDUCTIVE	PROPOSITION	
CHALLENGE	ENDIF	JUDGEMENT	RECURSIVE	
CLAIM	ENDTABLE	LAMBDA	SUBLEMMA	
CLOSURE	EXISTS	LAW	SUBTYPES	
COND	EXPORTING	LEMMA	SUBTYPE_OF	

C. An example of the generic OMDoc

```

<?xml version="1.0" encoding="UTF-8"?>
<omdoc id="swapelement.omdoc" catalogue="catalogue.omdoc">
  <metadata>
    <Creator role="aut">Xiao B. Jin</Creator>
    <Contributor role="ths">Doug. J. Howe</Contributor>
    <Date action="created">2004-07-15</Date>
  </metadata>
  <theory id="swapelement">
    <omtext>
      <CMP>procedure swapelement</CMP>
    </omtext>
    <symbol id="round_">
      <type system="sts">
        <OMOBJ>
          <OMA>
            <OMS cd="mathmltypes" name="fn_type" />
            <OMS cd="setname1" name="R" />
            <OMS cd="setname1" name="Z" />
          </OMA>
        </OMOBJ>
      </type>
    </symbol>
    <symbol id="atype" kind="type" scope="global">
      <type system="sts">
        <OMOBJ>
          <OMS cd="sts" name="type" />
        </OMOBJ>
      </type>
    </symbol>
    <definition id="atype" for="atype" type="simple">
      <FMP>
        <OMOBJ>
          <OMA>
            <OMS cd="mathmltypes" name="fn_type" />
            <OMS cd="setname1" name="Z" />
            <OMS cd="setname1" name="Z" />
          </OMA>
        </OMOBJ>
      </FMP>
    </definition>
    <symbol id="index" kind="type" scope="global">
      <type system="sts">
        <OMOBJ>
          <OMS cd="sts" name="type" />
        </OMOBJ>
      </type>
    </symbol>
    <definition id="index" for="index" type="simple">
      <FMP>
        <OMOBJ>
          <OMS cd="setname1" name="Z" />
        </OMOBJ>
      </FMP>
    </definition>
    <symbol id="index__last" kind="object" scope="global">
      <type system="sts">
        <OMOBJ>
          <OMS cd="setname1" name="Z" />
        </OMOBJ>
      </type>
    </symbol>
  </theory>
</omdoc>

```

```

</type>
</symbol>
<symbol id="index__first" kind="object" scope="global">
  <type system="sts">
    <OMOBJ>
      <OMS cd="setname1" name="Z" />
    </OMOBJ>
  </type>
</symbol>
<symbol id="integer__last" kind="object" scope="global">
  <type system="sts">
    <OMOBJ>
      <OMS cd="setname1" name="Z" />
    </OMOBJ>
  </type>
</symbol>
<symbol id="integer__first" kind="object" scope="global">
  <type system="sts">
    <OMOBJ>
      <OMS cd="setname1" name="Z" />
    </OMOBJ>
  </type>
</symbol>
<omtext id="var_a">
  <CMP>
    <OMOBJ>
      <OMS cd="swapelement" name="atype" />
      <OMV name="a" />
    </OMOBJ>
  </CMP>
</omtext>
<omtext id="var_j">
  <CMP>
    <OMOBJ>
      <OMS cd="setname1" name="Z" />
      <OMV name="j" />
    </OMOBJ>
  </CMP>
</omtext>
<omtext id="var_i">
  <CMP>
    <OMOBJ>
      <OMS cd="setname1" name="Z" />
      <OMV name="i" />
    </OMOBJ>
  </CMP>
</omtext>
<omtext>
  <CMP>rule_family swapelement_rules:</CMP>
</omtext>
<axiom id="swapelement_rules(3)">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="relation1" name="leq" />
        <OMS cd="swapelement" name="integer__first" />
        <OMS cd="swapelement" name="integer__last" />
      </OMA>
    </OMOBJ>
  </FMP>
</axiom>
<axiom id="swapelement_rules(7)">
  <FMP>
    <OMOBJ>

```

```

<OMA>
  <OMS cd="relation1" name="eq" />
  <OMS cd="swapelement" name="index__first" />
  <OMI>1</OMI>
</OMA>
</OMOBJ>
</FMP>
</axiom>
<axiom id="swapelement_rules(8)">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="relation1" name="eq" />
        <OMS cd="swapelement" name="index__last" />
        <OMI>10</OMI>
      </OMA>
    </OMOBJ>
  </FMP>
</axiom>
<assertion id="procedure_swapelement_1" type="theorem">
  <FMP>
    <OMOBJ>
      <OMBIND>
        <OMS cd="quant1" name="forall" />
        <OMBVAR>
          <OMATTR>
            <OMATP>
              <OMS cd="sts" name="type" />
              <OMS cd="setname1" name="Z" />
            </OMATP>
            <OMV name="i" />
          </OMATTR>
          <OMATTR>
            <OMATP>
              <OMS cd="sts" name="type" />
              <OMS cd="setname1" name="Z" />
            </OMATP>
            <OMV name="j" />
          </OMATTR>
          <OMATTR>
            <OMATP>
              <OMS cd="sts" name="type" />
              <OMS cd="swapelement" name="atype" />
            </OMATP>
            <OMV name="a" />
          </OMATTR>
        </OMBVAR>
      </OMBIND>
      <OMA>
        <OMS cd="logic1" name="and" />
        <OMS cd="logic1" name="true" />
        <OMA>
          <OMS cd="relation1" name="geq" />
          <OMV name="i" />
          <OMS cd="swapelement" name="index__first" />
        </OMA>
        <OMA>
          <OMS cd="relation1" name="leq" />
          <OMV name="i" />
          <OMS cd="swapelement" name="index__last" />
        </OMA>
        <OMA>
          <OMS cd="relation1" name="geq" />
          <OMV name="j" />
        </OMA>
      </OMA>
    </OMOBJ>
  </FMP>
</assertion>

```

```

    <OMS cd="swapelement" name="index__first" />
  </OMA>
<OMA>
  <OMS cd="relation1" name="leq" />
  <OMV name="j" />
  <OMS cd="swapelement" name="index__last" />
</OMA>
<OMBIND>
  <OMS cd="quant1" name="forall" />
  <OMBVAR>
    <OMATTR>
      <OMATP>
        <OMS cd="sts" name="type" />
        <OMS cd="type" name="index" />
      </OMATP>
      <OMV name="i__1" />
    </OMATTR>
  </OMBVAR>
<OMA>
  <OMS cd="logic1" name="implies" />
<OMA>
  <OMS cd="logic1" name="and" />
<OMA>
  <OMS cd="relation1" name="geq" />
  <OMV name="i__1" />
  <OMS cd="swapelement" name="index__first" />
</OMA>
<OMA>
  <OMS cd="relation1" name="leq" />
  <OMV name="i__1" />
  <OMS cd="swapelement" name="index__last" />
</OMA>
</OMA>
<OMA>
  <OMS cd="logic1" name="and" />
<OMA>
  <OMS cd="relation1" name="geq" />
<OMA>
  <OMS cd="fdl" name="element" />
  <OMV name="a" />
  <OMV name="i__1" />
</OMA>
  <OMS cd="swapelement" name="integer__first" />
</OMA>
<OMA>
  <OMS cd="relation1" name="leq" />
<OMA>
  <OMS cd="fdl" name="element" />
  <OMV name="a" />
  <OMV name="i__1" />
</OMA>
  <OMS cd="swapelement" name="integer__last" />
</OMA>
</OMA>
</OMBIND>
</OMA>
<OMA>
  <OMS cd="logic1" name="implies" />
<OMA>
  <OMS cd="logic1" name="and" />
<OMA>
  <OMS cd="relation1" name="eq" />
<OMA>

```

```

<OMS cd="fdl" name="update" />
<OMA>
  <OMS cd="fdl" name="update" />
  <OMV name="a" />
  <OMV name="i" />
  <OMA>
    <OMS cd="fdl" name="element" />
    <OMV name="a" />
    <OMV name="j" />
  </OMA>
</OMA>
<OMV name="j" />
<OMA>
  <OMS cd="fdl" name="element" />
  <OMV name="a" />
  <OMV name="i" />
</OMA>
</OMA>
<OMA>
  <OMS cd="fdl" name="update" />
  <OMA>
    <OMS cd="fdl" name="update" />
    <OMV name="a" />
    <OMV name="i" />
    <OMA>
      <OMS cd="fdl" name="element" />
      <OMV name="a" />
      <OMV name="j" />
    </OMA>
  </OMA>
  <OMV name="j" />
  <OMA>
    <OMS cd="fdl" name="element" />
    <OMV name="a" />
    <OMV name="i" />
  </OMA>
</OMA>
</OMA>
</OMA>
</OMA>
</OMOBJ>
</FMP>
</assertion>
</theory>
</omdoc>

```

D. An example of the PVS-oriented OMDoc

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE omdoc SYSTEM "../..//dtd/omdoc.dtd">
<omdoc id="swapelement.omdoc" catalogue="catalogue.omdoc">
  <metadata>
    <Creator role="aut">Xiao B. Jin</Creator>
    <Contributor role="ths">Doug. J. Howe</Contributor>
    <Date action="created">2004-07-15</Date>
  </metadata>
  <theory id="swapelement">
    <omtext>
      <CMP>define fdl mod function using PVS rem funnction</CMP>
    </omtext>
    <symbol id="mod" kind="object" scope="global">
      <type system="pvs">
        <OMOBJ>
          <OMA>
            <OMS cd="pvs" name="funtype" />
            <OMS cd="integers" name="int" />
            <OMS cd="integers" name="int" />
            <OMS cd="integers" name="int" />
          </OMA>
        </OMOBJ>
      </type>
    </symbol>
    <definition id="mod" for="mod" type="simple">
      <FMP>
        <OMOBJ>
          <OMA>
            <OMS cd="equalities" name="=" />
            <OMS cd="pvs" name="mod" />
            <OMBIND>
              <OMS cd="pvs" name="lambda" />
            <OMBVAR>
              <OMATTR>
                <OMATP>
                  <OMS cd="pvs" name="type" />
                  <OMS cd="integers" name="int" />
                </OMATP>
                <OMV name="x" />
              </OMATTR>
              <OMATTR>
                <OMATP>
                  <OMS cd="pvs" name="type" />
                  <OMS cd="integers" name="int" />
                </OMATP>
                <OMV name="y" />
              </OMATTR>
            </OMBVAR>
          </OMA>
          <OMA>
            <OMS cd="pvs" name="rem" />
          </OMA>
          <OMS cd="real_defs" name="abs" />
          <OMV name="x" />
        </OMOBJ>
      </FMP>
    </definition>
  </theory>
</omdoc>

```

```

        <OMS cd="real_defs" name="abs" />
    <OMV name="y" />
  </OMA>
</OMA>
</OMBIND>
</OMA>
</OMOBJ>
</FMP>
</definition>
<omtext>
  <CMP>procedure swapelement</CMP>
</omtext>
<symbol id="round_">
  <type system="pvs">
    <OMOBJ>
      <OMA>
        <OMS cd="pvs" name="funtype" />
        <OMS cd="reals" name="real" />
        <OMS cd="integers" name="int" />
      </OMA>
    </OMOBJ>
  </type>
</symbol>
<symbol id="atype" kind="type" scope="global">
  <type system="pvs">
    <OMOBJ>
      <OMS cd="pvs" name="type" />
    </OMOBJ>
  </type>
</symbol>
<definition id="atype" for="atype" type="simple">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="pvs" name="funtype" />
        <OMS cd="integers" name="int" />
        <OMS cd="integers" name="int" />
      </OMA>
    </OMOBJ>
  </FMP>
</definition>
<symbol id="index" kind="type" scope="global">
  <type system="pvs">
    <OMOBJ>
      <OMS cd="pvs" name="type" />
    </OMOBJ>
  </type>
</symbol>
<definition id="index" for="index" type="simple">
  <FMP>
    <OMOBJ>
      <OMS cd="integers" name="int" />
    </OMOBJ>
  </FMP>
</definition>
<symbol id="index__last" kind="object" scope="global">
  <type system="pvs">
    <OMOBJ>
      <OMS cd="integers" name="int" />
    </OMOBJ>
  </type>
</symbol>
<symbol id="index__first" kind="object" scope="global">
  <type system="pvs">

```

```

    <OMOBJ>
      <OMS cd="integers" name="int" />
    </OMOBJ>
  </type>
</symbol>
<symbol id="integer__last" kind="object" scope="global">
  <type system="pvs">
    <OMOBJ>
      <OMS cd="integers" name="int" />
    </OMOBJ>
  </type>
</symbol>
<symbol id="integer__first" kind="object" scope="global">
  <type system="pvs">
    <OMOBJ>
      <OMS cd="integers" name="int" />
    </OMOBJ>
  </type>
</symbol>
<omtext id="var_a">
  <CMP>
    <OMOBJ>
      <OMS cd="swapelement" name="atype" />
      <OMV name="a" />
    </OMOBJ>
  </CMP>
</omtext>
<omtext id="var_j">
  <CMP>
    <OMOBJ>
      <OMS cd="integers" name="int" />
      <OMV name="j" />
    </OMOBJ>
  </CMP>
</omtext>
<omtext id="var_i">
  <CMP>
    <OMOBJ>
      <OMS cd="integers" name="int" />
      <OMV name="i" />
    </OMOBJ>
  </CMP>
</omtext>
<omtext>
  <CMP>rule_family swapelement_rules:</CMP>
</omtext>
<axiom id="swapelement_rules(3)">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="reals" name="<=" />
        <OMS cd="swapelement" name="integer__first" />
        <OMS cd="swapelement" name="integer__last" />
      </OMA>
    </OMOBJ>
  </FMP>
</axiom>
<axiom id="swapelement_rules(7)">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="equalities" name="=" />
        <OMS cd="swapelement" name="index__first" />
      </OMA>
    </OMOBJ>
  </FMP>
  <OMI>1</OMI>

```



```

</OMA>
</OMOBJ>
</FMP>
</axiom>
<axiom id="swapelement_rules(8)">
  <FMP>
    <OMOBJ>
      <OMA>
        <OMS cd="equalities" name="=" />
        <OMS cd="swapelement" name="index__last" />
        <OMI>10</OMI>
      </OMA>
    </OMOBJ>
  </FMP>
</axiom>
<assertion id="procedure_swapelement_1" type="theorem">
  <FMP>
    <OMOBJ>
      <OMBIND>
        <OMS cd="pvs" name="forall" />
        <OMBVAR>
          <OMATTR>
            <OMATP>
              <OMS cd="pvs" name="type" />
              <OMS cd="integers" name="int" />
            </OMATP>
            <OMV name="i" />
          </OMATTR>
          <OMATTR>
            <OMATP>
              <OMS cd="pvs" name="type" />
              <OMS cd="integers" name="int" />
            </OMATP>
            <OMV name="j" />
          </OMATTR>
          <OMATTR>
            <OMATP>
              <OMS cd="pvs" name="type" />
              <OMS cd="swapelement" name="atype" />
            </OMATP>
            <OMV name="a" />
          </OMATTR>
        </OMBVAR>
      </OMBIND>
      <OMA>
        <OMS cd="booleans" name="AND" />
        <OMS cd="booleans" name="TRUE" />
        <OMA>
          <OMS cd="reals" name="&gt;=" />
          <OMV name="i" />
          <OMS cd="swapelement" name="index__first" />
        </OMA>
        <OMA>
          <OMS cd="reals" name="&lt;=" />
          <OMV name="i" />
          <OMS cd="swapelement" name="index__last" />
        </OMA>
        <OMA>
          <OMS cd="reals" name="&gt;=" />
          <OMV name="j" />
          <OMS cd="swapelement" name="index__first" />
        </OMA>
        <OMA>
          <OMS cd="reals" name="&lt;=" />

```

```

<OMV name="j" />
<OMS cd="swapelement" name="index__last" />
</OMA>
<OMBIND>
<OMS cd="pvs" name="forall" />
<OMBVAR>
<OMATTR>
<OMATP>
<OMS cd="pvs" name="type" />
<OMS cd="swapelement" name="index" />
</OMATP>
<OMV name="i__1" />
</OMATTR>
</OMBVAR>
<OMA>
<OMS cd="booleans" name="IMPLIES" />
<OMA>
<OMS cd="booleans" name="AND" />
<OMA>
<OMS cd="reals" name="&gt;=" />
<OMV name="i__1" />
<OMS cd="swapelement" name="index__first" />
</OMA>
<OMA>
<OMS cd="reals" name="&lt;=" />
<OMV name="i__1" />
<OMS cd="swapelement" name="index__last" />
</OMA>
</OMA>
<OMA>
<OMS cd="booleans" name="AND" />
<OMA>
<OMS cd="reals" name="&gt;=" />
<OMA>
<OMS cd="pvs" name="element" />
<OMV name="a" />
<OMV name="i__1" />
</OMA>
<OMS cd="swapelement" name="integer__first" />
</OMA>
<OMA>
<OMS cd="reals" name="&lt;=" />
<OMA>
<OMS cd="pvs" name="element" />
<OMV name="a" />
<OMV name="i__1" />
</OMA>
<OMS cd="swapelement" name="integer__last" />
</OMA>
</OMA>
</OMA>
</OMBIND>
</OMA>
<OMA>
<OMS cd="booleans" name="IMPLIES" />
<OMA>
<OMS cd="booleans" name="AND" />
<OMA>
<OMS cd="equalities" name="=" />
<OMA>
<OMS cd="pvs" name="updates" />
<OMA>
<OMS cd="pvs" name="updates" />
<OMV name="a" />

```

```

<OMV name="i" />
<OMA>
  <OMS cd="pvs" name="element" />
  <OMV name="a" />
  <OMV name="j" />
</OMA>
</OMA>
<OMV name="j" />
<OMA>
  <OMS cd="pvs" name="element" />
  <OMV name="a" />
  <OMV name="i" />
</OMA>
</OMA>
<OMA>
  <OMS cd="pvs" name="updates" />
  <OMA>
    <OMS cd="pvs" name="updates" />
    <OMV name="a" />
    <OMV name="i" />
  </OMA>
  <OMS cd="pvs" name="element" />
  <OMV name="a" />
  <OMV name="j" />
</OMA>
</OMA>
<OMV name="j" />
<OMA>
  <OMS cd="pvs" name="element" />
  <OMV name="a" />
  <OMV name="i" />
</OMA>
</OMA>
</OMA>
</OMA>
</OMOBJ>
</FMP>
</assertion>
</theory>
</omdoc>

```