Master Thesis
Technical Information Technology / Software Engineering

Coupling Overture to MDA and UML

by

Kenneth G. Lausdahl
Hans Kristian A. Lintrup

Supervisor: Peter Gorm Larsen

Aarhus, December 2008
# Contents

1. **Introduction**
   1.1 The formal method ........................................ 9
   1.2 The informal method ....................................... 10
   1.3 Model transformation ..................................... 10
   1.4 Participation in the Overture Project .................... 11
   1.5 Thesis Goal ............................................. 12
   1.6 Reading Guidelines ...................................... 13
   1.7 Related Work ............................................ 14
   1.8 Outline of the Thesis .................................... 15
     1.8.1 Quick Overview of the Thesis ......................... 16
     1.8.2 Thorough Exposition of the Thesis .................. 16

2. **UML**
   2.1 History of UML ........................................ 19
   2.2 UML Usage ............................................. 20
     2.2.1 KLEIN+STEKL GmbH .................................. 20
     2.2.2 Borland Together Control Center .................... 20
     2.2.3 Telelogic Rhapsody .................................. 21
   2.3 The UML Meta-model ................................... 21
   2.4 Choosing versions of UML for comparison ............... 23
   2.5 UML 1 ................................................ 23
     2.5.1 Class Diagram ...................................... 24
     2.5.2 Sequence Diagram ................................... 29
     2.5.3 Rose-VDM++ Link .................................... 31
   2.6 UML 2 ................................................ 36
     2.6.1 Class Diagram ...................................... 36
     2.6.2 Sequence Diagram ................................... 37
   2.7 Tool support for XML Metadata Interchange (XMI) ....... 40
     2.7.1 XMI incompatibilities between UML modeling tools . 40
     2.7.2 Limitation of Sequence Diagrams in XMI ............. 41
## 3 VDM

3.1 History of VDM ................................................. 43
3.2 VDM Usage ......................................................... 44
    3.2.1 Banknote Processing ..................................... 44
    3.2.2 VDMTools .................................................. 45
    3.2.3 ISEPUMS .................................................... 45
    3.2.4 A Mission Critical Data Handling Subsystem .............. 45
3.3 Tool support ..................................................... 46
3.4 VDM Classes ..................................................... 47
3.5 Types ........................................................... 48
3.6 Test Trace ....................................................... 49
3.7 Trace Example ................................................... 52

## 4 Static Model Transformation

4.1 Classes ......................................................... 53
4.2 Visibility ....................................................... 53
4.3 Data types ...................................................... 54
4.4 Instance variables and values ................................ 55
4.5 Union Types ...................................................... 57
4.6 Product Types ................................................... 57
4.7 Collections ...................................................... 58
4.8 Relationships .................................................... 61
4.9 Thread .......................................................... 62
4.10 Generalization .................................................. 63
4.11 Abstract ......................................................... 63
4.12 Generic classes ................................................. 64
4.13 Operations and Functions ..................................... 65

## 5 Static Model Specification

5.1 Abstract Syntax Tree ............................................. 67
    5.1.1 AST definition to VDM class structure (ASTGen) ......... 67
    5.1.2 OML AST ................................................... 70
    5.1.3 UML AST .................................................. 71
5.2 Transformation Specification Overview ....................... 72
5.3 The Transformation Process ................................... 73
5.4 Transforming VDM to UML ..................................... 75
5.5 UML Model to XMI .............................................. 82
    5.5.1 XML parser / deparser ..................................... 84
5.6 Transforming UML to VDM ..................................... 84
5.7 Merging Changes in VDM and UML Models ................... 87

## 6 Interaction Model Transformation

................................................................. 89
### CONTENTS

6.1 VDM traces and UML sequence diagrams ........................................ 90
6.2 Transformation Rules ................................................................. 91
   6.2.1 Trace placement ................................................................. 91
   6.2.2 Trace name ................................................................. 93
   6.2.3 Trace Apply Expression ..................................................... 93
   6.2.4 Sequencing of trace apply expressions ................................ 94
   6.2.5 Trace choice operator ...................................................... 95
   6.2.6 Repeat Pattern for apply expressions .................................. 95
   6.2.7 Nested sequencing messages ............................................. 96

7 Interaction Model Specification ......................................................... 99
   7.1 Subset of UML AST in relation to sequence diagrams .................. 99
   7.2 Transformation Specification Overview .................................. 100
   7.3 Transforming UML SD to VDM Trace ........................................ 102
      7.3.1 Summary of traces specification .................................... 108

8 Transformation implementation ......................................................... 109
   8.1 Testing ................................................................................. 109
      8.1.1 Script testing ............................................................. 110
      8.1.2 Unit test .................................................................... 110
   8.2 Java code-generator for VDM ................................................... 111
   8.3 Integrating UML in Overture Tool ............................................ 111
      8.3.1 Development of the UML Plug-in .................................... 113
      8.3.2 Deployment of the Plug-in ............................................. 113

9 Concluding Remarks ........................................................................ 115
   9.1 Achieved Results .................................................................... 115
      9.1.1 Learning outcome ...................................................... 115
      9.1.2 Concrete achievements ................................................ 116
   9.2 Future Work .......................................................................... 119
   9.3 Overall Conclusion .................................................................. 120

A Overture Workshop 5 in Braga Portugal ............................................ 133
   A.1 Participation in Workshop ...................................................... 133
   A.2 What did we gain from the Workshop ..................................... 133
   A.3 Workshop conclusion .......................................................... 134

B Omitted UML 1 Constructs ............................................................... 135
   B.1 Association ........................................................................ 135
   B.2 Dependency ........................................................................ 136
   B.3 Derived Element .................................................................. 136
   B.4 Package and subsystem ...................................................... 136
   B.5 Association Class ............................................................. 136
CONTENTS

B.6 Interface .................................................. 137
B.7 Realization ................................................. 138
B.8 Attributes of metaclass Class .......................... 138
B.9 Concurrency ............................................... 139
B.10 DataType .................................................. 139

C Omitted UML 2 Constructs ............................... 141
C.1 Internal Structure of a Class ......................... 141
C.2 Message kind .............................................. 141
   C.2.1 Part decomposition .................................. 142
C.3 Fragment .................................................. 143
   C.3.1 ConsiderIgnore Fragment .......................... 144
   C.3.2 InteractionUse ...................................... 145

D Significant changes to the UML meta-model .......... 147
D.1 Deprecated UML 1 meta-classes ..................... 147
D.2 New UML 2 meta-classes ............................... 148

E Specification of the UML Abstract Representation .... 149
E.1 Class Diagram ............................................. 150
   E.1.1 Model and ModelElement ......................... 150
   E.1.2 Class ................................................. 150
   E.1.3 Type ................................................ 153
   E.1.4 Association ......................................... 154
   E.1.5 Constraint and ValueSpecification ............. 154
E.2 Sequence Diagram ....................................... 155
   E.2.1 Collaboration ....................................... 155
   E.2.2 LifeLine ............................................. 156
   E.2.3 InteractionFragment ............................... 156
   E.2.4 Message ............................................. 159
E.3 UML Specification Citations ......................... 160

F Model coverage ............................................. 163
F.1 Transforming from VDM to UML ...................... 163
   F.1.1 Transformation from VDM to UML (Vdm2Uml) .......... 163
   F.1.2 VDM to UML type converter (Vdm2UmlType) ........ 178
   F.1.3 Serilize the UML AST to XMI with EA support (Uml2XmiEAxml) 182
F.2 Transforming UML to VDM ............................. 199
   F.2.1 Convert XMI to a UML model (Xml2UmlModel) ........ 199
   F.2.2 Transform UML to VDM (Uml2Vdm) .................. 215
F.3 OML AST to VDM files printer ...................... 232
   F.3.1 Proxy for printer (Oml2Vpp) ...................... 232
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F.3.2 Visitor for OML which implements a printer for source files (Oml2VppVisitor)</td>
<td>233</td>
</tr>
<tr>
<td>G    OML AST</td>
<td>253</td>
</tr>
<tr>
<td>H    UML AST</td>
<td>277</td>
</tr>
<tr>
<td>I    Features supported by Transformation</td>
<td>283</td>
</tr>
<tr>
<td>List of Symbols and Abbreviations</td>
<td>285</td>
</tr>
</tbody>
</table>
Abstract

It is vital that critical software systems perform as intended. An effective way to minimize the risk of unforeseen surprises in a system is to create a model of the system’s critical parts. VDM++ is an OO modeling language used to validate and verify the design of software systems at a desired level of abstraction. VDMTools is a toolset which offers various features to support software development based on VDM and its predecessor, VDM-SL. Among the features offered by VDMTools, is the Rose-VDM++ Link which enables going back and forth between VDM++ model and UML version 1.1. This M.Sc. thesis presents an analysis of the additional possibilities offered by UML version 2. The results of this are materialized as an extension of the Overture project which is a community-based project dedicated to the development of the next generation of tools supporting formal modeling and analysis in the design of software systems. We have developed a counterpart to the Rose-VDM++ Link, i.e. a tool for going back and forth between VDM++ models and UML 2.0 Class Diagrams and Sequence Diagrams.
Resumé

Acknowledgements

First of all, we would like to thank our supervisor, Professor Peter Gorm Larsen, for the professional and personal support during this thesis work. His dedication is remarkable and has resulted in correspondence both day and night regarding this thesis. At no point during the writing of this thesis has Peter been absent when needed. His support and guidance is second to none, both personally and professionally. Secondly, a warm thank you goes to Nokia, Sony/FeliCa Networks and Shin Sahara who sponsored our travel expenses, which enabled both of us to participate in the fifth Overture Workshop in Braga, Portugal. Thank you very much for that. Thirdly, Marcel Verhoef deserves thanks for sharing his knowledge regarding the Overture project and its tools. Fourthly, we would like to express our appreciation to Nick Battle for his continuous interest in our thesis work regarding VDMJ. Nick was extremely quick to resolve the deficiencies we discovered. We have not had direct relations to the last person whom we want to thank, however, he has been a valuable help during the construction of our VDM++ model. When we encountered a bug in VDMTools, we submitted a bug-report. Shin Sahara heads the development of VDMTools and one of his staff members, Dr. K, is the person who has agreed to fix a number of the bugs encountered by us.
Prerequisites

It is expected that the reader is able to read basic formal models written in VDM++ and possess base knowledge about UML class diagrams and sequence diagrams. For simplicity, we refer to VDM++ as VDM in the rest of this thesis.
Chapter 1

Introduction

Between 1985 and 1987, the radiation therapy machine Therac-25 was responsible for at least six accidents in which patients were given massive overdoses of radiation, approximately 100 times the intended dose. Three of the six patients died [Therac25].

Such an accident highlights the danger of software control of safety-critical systems. Formal methods (FM) mitigate the risk of a system malfunction, like Therac-25, by increasing the confidence in a computer system by formal verification of the systems specification. Hence it is interesting to investigate how to spread the use of FM.

This thesis investigates the possibility of combining an FM with an informal graphical modeling language. The topic is well-known in the literature and is referred to as model transformation [Kim&05, Snook&06, Dascalu&02, Laleau00].

The use of FM stem from the fact that software is notorious for being late in delivery and unpredictable and unreliable in operation. The expectation is that proper mathematical analysis can contribute to the reliability and robustness of a design [Holloway97]. However, developers are reluctant to devote themselves to FM. Suggestions listed by Holloway regarding the antipathy against FM includes, but are not limited to, the following [Holloway97, p1]:

1. Lack of adequate tools [Knight&97, p1].
2. High costs, and over-selling by advocates [Meyer97, p1] [Saiedian96, p2].

It is the hope, that this thesis work will contribute to the spread of FM by mitigating the two causes mentioned above. In particular, the ambition is that tool support for a specific FM will be improved with reduced cost, as a result of this work.

1.1 The formal method

The FM chosen for this thesis is the Vienna Development Method extended with object-oriented capabilities (VDM++) [Fitzgerald&05]. VDM++ is one of the oldest and most
mature formal languages available [Fitzgerald&08a, Plat&92]. VDM++ is an extension of the specification language VDM-SL, thus a VDM++ model can be formally validated and verified by using a set of tools [Fitzgerald&08a]. A VDM++ model specifies the behavior of a system by encapsulating system behavior in classes, as known from the object-oriented world. Recently, a new construct for improving model testing has been introduced. This new feature called trace statements makes it easier to specify regression tests [Santos08, LangManPPTraces]. The new traces feature, together with VDM++, are interesting to take into account when performing a transformation to and from an informal method.

1.2 The informal method

The chosen graphical modeling language is the Unified Modeling Language (UML) [UMLSuperstructure2.1.2]. UML is widely used in the industry [UMLSuccess].

UML is great for presenting and discussing models due to its visual capabilities, i.e. different structural and behavioral views of a system. Of the different views, Class Diagrams and Sequence Diagrams are of particular interest to our work. One of the main disadvantages, however, is the level of formalism that can be obtained, since UML has no commonly agreed mathematical basis. However, the reason for that is the significant complexity it would add with no clear benefit [UMLInfrastructure2.1.2, p33]. VDM++ has a level of mathematical precision but it is not great for presenting or discussing a model to computer science novices, e.g. management or customers, or to developers unfamiliar with FM. A link between VDM++ and UML yields the best of both worlds [Dascalu&02, Kim&05], i.e. the formal specification of VDM++ becomes easier to apply and the informal language UML becomes more precise [Dascalu&02, Kim&05].

UML is designed to model object-oriented systems [UMLFromWikipedia], hence overall similarities do exist between VDM++ and UML. A careful analysis of the languages will determine the mapping potential between them.

1.3 Model transformation

If a mapping potential exists between VDM++ and UML, detailed knowledge of the semantics of the languages involved must be obtained in order to know:

- On what semantic basis the transformation has occurred
- Whether semantics are preserved during a transformation (soundness), and
- whether the transformation is complete [Kim&05, Sendall&03].

[Sendall&03] identifies meta-modeling as a common technique for defining the abstract syntax of models and the relationships between model elements. This thesis adopts that approach. The abstract syntax for VDM++ was available by the open-source project
1.4. Participation in the Overture Project

Overture [OvertureTool] at the time this thesis work began. The definition of an abstract syntax for UML is made as a part of this thesis.

Once the abstract syntaxes exist at the same level of abstraction, it is possible to define transformation rules for the meta-model constructs of each language. The rules are first formulated in natural language, and secondly specified in a formal language in order to be validated and verified. Ultimately, the rules must be implemented and compiled to some executable form in order to act as a tool.

In this regard, VDM++ is interesting because VDM++ models may be executed and debugged directly on the specification level [Fitzgerald&08a]. A VDM++ model of the model transformation will allow developers to abstract away parts not directly related to the core functionality. Also, a VDM++ model may be subject to syntax checking, type checking and integrity checking to increase confidence in the correctness of the model [Fitzgerald&08a].

1.4 Participation in the Overture Project

The Overture project is an open source project led by a core team, who discusses and plans development of the Overture project. The aim is to enable better tool support for VDM++. Currently, the most feature-rich tool available is a commercial tool, VDMTools, which include features like syntax- and type-check, code-generation etc.

A model has been created to enable transformation between the abstract syntax of VDM++ and the abstract syntax of UML. The model utilizes a variety of VDM++ constructs and due to the large amount of classes, VDMTools becomes significantly slow when interpreting the model. This brought to our attention VDMJ [Fujitsu] developed by Nick Battle at Fujitsu. VDMJ is a console-based type-checker and interpreter which is roughly twice as fast as VDMTools. During the project close cooperation with Nick Battle has been maintained to test and further develop VDMJ. Additionally, to the development of VDMJ, we participated in the Overture project by means of net-meetings and we attended the fifth Overture Workshop at the University of Minho in Braga, Portugal.

One of the main topics at the workshop was integrating various existing tools into a single workspace and to discuss how to integrate these tools in a single Eclipse platform. The solution for arranging all existing tools was to use Maven. However, Maven did not know how VDMTools projects are structured, hence a VDMTools plug-in for Maven was required.

We contributed with a presentation of the challenges and expected outcome of this thesis and we started the development of a VDMTools plug-in for Maven, which provided a solid starting point for further development.
1.5 Thesis Goal

The primary goal of this thesis is to investigate the mapping potential between VDM++ and UML. More specifically, this thesis will uncover the mapping potential between:

- VDM++ models and UML 2 Class Diagrams, and
- VDM++ traces and UML 2 Sequence Diagrams

To accomplish the abovementioned, it is necessary to conduct a syntax and semantics analysis of both VDM++ and UML. The analysis will determine a number of language constructs which can be transformed with their semantics maintained. The second goal of this thesis is to formulate bidirectional transformation rules for each identified language construct. The rules must be stated in natural language and subsequently be defined formally using VDM++. Once the rules have been formally defined, the ambition is to develop a prototype of a transformation tool incorporating the formally defined rules.

The output of the prototype must comply with the concrete syntaxes of VDM++ and UML, to enable importing the output into existing tools (i.e. an existing UML tool should be able to import the output of a VDM++ to UML transformation). To accomplish that regarding UML, it is necessary to first investigate the existing standard for UML diagram exchange by the Object Management Group (OMG), and secondly to explore to which degree various UML tool vendors adhere to the standard. In addition, the concrete syntax of VDM++ must be examined to produce correctly formatted output.

To summarize, the subgoals of this thesis are:

**UML 1 and UML 2:** To investigate the UML specifications, i.e. syntax and semantics of UML 1 and UML 2.

**VDM++:** To examine the VDM++ syntax and semantics, including the new traces definitions [Santos08].

**Mapping potential:** To determine the mapping potential between VDM++ and UML in terms of language constructs which can maintain their semantics during a transformation.

**Transformation rules:** To formulate in natural language a collection of bidirectional transformation rules for each language construct, i.e. one rule for each direction (VDM++ and UML roundtrip). The rules must subsequently be formally specified in VDM++.

**Diagram exchange standard:** To investigate the OMG standard for diagram exchange in order to represent a UML model correctly.

**Prototype:** To develop a model of the model transformation using VDM++. The prototype will assist to ascertain the level of correctness of the transformation rules.
In addition, the prototype will assist in uncovering whether existing UML tools adhere to the diagram exchange standard.

1.6 Reading Guidelines

In order to ease the reading process commonly used illustration principles are shown below along with a short description of what they present. In addition, please notice that the page numbers given in citations of articles, correspond to pages.

Below is a description of how different types of models and source code are displayed and how names or keywords from within this models/source illustrations are shown:

Keyword: bool are shown in boldface.

References to names in models/source code are shown without boldface getValue.

Representation box of a VDM model:

```
| class SensorController |
| thread |
| while true do |
| skip; |
| end SensorController |
```

Listing 1.1: Example of VDM.

Representation box of an AST:

```
TemplateParameter :::
    name : seq of char;
```

Listing 1.2: Example of an AST

Representation box of a XML files:

```
<xml:Extension extender="Enterprise Architect" extenderID="6.5"> 
</xml:Extension>
```

Listing 1.3: Example of XMI/XML.

Representation of a transformation rule:

**Transformation Rule 1**

VDM classes are mapped as the UML meta-class Class
CHAPTER 1. INTRODUCTION

Connection between two items at different levels:

```java
class SensorController
thread
while true do
   skip;
end SensorController
```

Listing 1.4: A VDM class with a thread definition.

1.7 Related Work

The idea of combining formal and informal languages to exploit the best of both worlds [Dascalu&02, Kim&05, RoseMan] have been investigated by others before. Common to [Kim&05, Snook&06, Dascalu&02, Laleau00, RoseMan] is the mapping between a formal method (Alloy, Z, Z++, VDM++) an the UML Class Diagrams which provides a static view. Not only UML Class Diagrams have been considered for transforming between formal and informal models. UML Sequence diagrams [Dascalu&02] and state machines [Guelfi&08] have also been subject of model transformations.

A To enable such a transformation, rules must be stated in order to before a actual transformation can take place. The specification of such rules are critical and often not explicitly defined [Kim&05]. These rules can be specified in a formal way and proved as in [Laleau00], where they proved 80 % of the rules specified in B by isabelle/HOL.

[McUmber&01] have formalized UML in terms of Promela, the formal system description language for SPIN\(^1\). The mapping process from UML to a target language has been automated in a tool called Hydra, which is a prototype to demonstrate that the rules are sufficiently defined that formal language specifications can be generated automatically from UML diagrams.

[Konrad&05] describe a tool for specifying and analyzing natural language properties of UML models, resulting in generation of the corresponding formal specification language Promela, which can then be formally analyzed by the model checker Spin. Several tools for the behavioral analysis of UML models have been developed, where a user typically specifies properties in terms of formal specification languages. The aim of their tool is to ease the use of formal specification languages by being able to accept natural language as input. Natural language properties are derived using a grammar that supports certain specification patterns. Their grammar supports the natural language representation of these specification patterns. The grammar is used to specify linear-time temporal logic (LTL) properties, i.e. encode formulae about a condition which will eventually be true. The grammar can be customized according to vocabulary and speci-

\(^1\)Used for formal verification of distributed software systems.
1.8. OUTLINE OF THE THESIS

This thesis is structured into 9 chapters, each covering a particular topic of the model transformation. The following section gives a quick overview of the thesis. The succeeding section gives a more elaborate account of the chapters of this thesis.
CHAPTER 1. INTRODUCTION

1.8.1 Quick Overview of the Thesis

The first part of this thesis introduces the reader to UML and VDM++. The next two parts are concerned with the model transformations between VDM++ and UML Class Diagrams and VDM++ traces and UML Sequence Diagrams, in terms of transformation rules and how they are specified using VDM++. Then follows a discussion of the implementation of the transformation rules the use of various tools to achieve the model transformation. Finally we conclude on the work presented in this thesis.

1.8.2 Thorough Exposition of the Thesis

Chapter 2 gives an introduction of the informal language UML regarding its history and general usage. The chapter then focuses on the syntax and semantics of UML. It ends with an introduction of XML Metadata Interchange (XMI), which is the standard for diagram exchange advocated by OMG.

Chapter 3 introduces the formal language VDM++ and its uses in the industry. Next, a description of the syntax and semantics of VDM++ is given, supplemented with an account of current tool support. The chapter ends with a description of the new concept of VDM traces.

After the introduction of the two modeling languages, Chapter 4 gives the reader a thorough explanation of the bidirectional transformation rules, which enable going back and forth between a VDM++ model and a UML model.

Chapter 5 follows up on the preceding chapter by giving a description of how the transformation rules from Chapter 4 are turned into a VDM model. The chapter also introduces the reader to Abstract Syntax Trees, the Overture tool ASTGen and how to merge changes between VDM++ and UML models.

The thesis now turns the attention towards how to perform a model transformation between VDM++ and UML Sequence Diagrams. Chapter 6 first discusses which combinations of VDM++ traces and UML Sequences Diagrams yield the greatest benefit. Then follows a description of how VDM++ traces are related to UML 2 Sequence Diagrams in terms of transformation rules.

Chapter 7 describes how the transformation rules from Chapter 6 are added to the existing model transformation described in Chapter 5.

At this point in time, the reader will have knowledge of the structure of both VDM++ and UML in addition to a complete set of transformation rules between the two languages. Chapter 8 introduce the reader to the implementation of the model transformation. The
implementation is facilitated tool-based code-generation to Java.

Finally, in Chapter 9, we conclude on the thesis work. Each subsection of the conclusion treat a certain angle of this thesis work. The thesis is ends with an overall conclusion which sum up relevant aspects of the thesis in its entirety.

Appendix A gives an outline important aspects regarding our participation in the Fifth Overture Workshop in Braga, Portugal.

Appendix B contains a description of omitted UML 1 constructs, i.e. the constructs which have no VDM++ counterpart.

Appendix C contains a description of omitted UML 2 constructs.

Appendix D contains a description of the changes to the UML meta-model from UML 1 to UML 2.

Appendix E contains the specification of the UML AST, i.e. the syntactical description of the UML abstract syntax.

Appendix F contains the model coverage, i.e. it shows which lines of the model are being used during a model transformation.

Appendix G contains the entire OML AST as provided by the Overture project [Overture07].

Appendix I presents an overview of the extent to which the model transformation is implemented.

A list of symbols and abbreviations can be found in the back thesis.
Chapter 2

UML

This chapter starts by a short introduction to the history and general usage of the Unified Modeling Language (UML). Then a brief overview of the structure of UML follows to prepare the reader for the sections describing the constructs of UML 1.4.2 (denoted UML 1) and UML 2.1.2 (denoted UML 2) and the differences between the two versions. The chapter ends with a description of the XML Metadata Interchange (XMI) format by OMG to enable interchange of diagram instance data among different tool vendors.

2.1 History of UML

In the mid-1970s and the late 1980s various object-oriented (OO) modeling methodologies began to appear as a consequence of the emerging OO analysis and design. More than fifty different of modeling languages occurred in the period between 1989-1994 and developers had difficulties finding a methodology that satisfied their needs [UML 1.4.2, p33-34]. In the mid-1990s the creators of two leading methods, the Object Modeling Technique (OMT) and the Booch Method by Rumbaugh and Booch, respectively, began to assimilate from other methods what they considered to be of interest [UML Distilled, p7-8]. Together Rumbaugh and Booch attempted to reconcile their two approaches and started to work on a Unified Method [UML Distilled, p7-8]. In 1993 they were accompanied by Jacobson who brought with him the object-oriented software engineering (OOSE) method which was a use-case oriented approach that provided excellent support for business engineering and requirements analysis [UML Distilled, p7-8].

Rumbaugh, Booch and Jacobson was summoned in 1996 by Rational to head the development of a non-proprietary Unified Modeling Language that should be presented to the Object Management Group (OMG) for adoption [UML Distilled, p7-8]. The rationale was that the abundance of methodologies was impeding the spreading of the OO approach and a unified language would help settle the differences [UML Distilled, p7-8]. The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9
and 0.91 in June and October of 1996 [UML1.4.2, p33-34]. In 1997 Rational released version 1.0 of the UML documentation as their proposal to OMG. The proposal included suggestions from various organisations which was merged and the resulting version 1.1 was adopted by OMG [UMLDistilled, p8].

In January 2005 the International Standardization Organization (ISO) released version 1.4.2 of UML as an international standard (ISO/IEC 19501) and in July 2005 OMG released UML 2.0 which presented the most radical changes to the UML since version 1.1 [OMGUMLHomepage], [UMLDistilled, p157]. The present version of UML at the time of this writing is 2.1.2 [OMGUMLHomepage].

2.2 UML Usage

UML is a semi-formal visual modeling language used by developers to model a system at a desired level of abstraction. It allows developers to step back and look at a system, or a subpart thereof, from a more general point of view. The different views comprise thirteen kinds of diagrams distributed between two primary categories: structure and behavior [UMLSuperstructure2.1.2, p700]. This thesis focuses on Class Diagrams (static structure) and Sequence Diagrams (behavior). Several examples of the industrial use of UML exist [UMLSuccess], some of which are described below.

2.2.1 KLEIN+STEKL GmbH

The German software company KLEIN+STEKL GmbH provided a solution to the Operations Department of Zuercher Kantonalbank (ZKB), one of the three leading banks in Switzerland. The system serves about 60 NT-clients. The users are dealing with about 330 different data classes and about 220,000 objects saved in the database. The SDV Tool\(^1\) consists of some 109,000 Java statements in 1,335 classes. The whole system was designed with UML (Rational Rose [RationalRose]). As UML is easy to understand, use cases, data models and interactions can be discussed even with the end users, thus ensuring a practical solution [ZurcherKantolbank].

2.2.2 Borland Together Control Center

The Charles Schwab Corporation provides securities brokerage and related financial services for 8 million active accounts with $837 billion in assets. Schwab senior technology management recognized the need to facilitate the consistency of architecture and development models across multiple projects, and get developers speaking the same language. After surveying its developers, and comparing features to features, Together Control Center was selected. One of the biggest benefits to the developers at Schwab is the Together Control Center reverse engineering feature, which takes existing code and allows the developer to visualize the model using UML. Another important feature

\(^1\)Tool for Master Data Management (in-house product.)
to the developers is simultaneous round-trip engineering, which ensures instantaneous
code and model matching allowing for fewer bugs and faster testing. [Schwab]

2.2.3 Telelogic Rhapsody

Telelogic Rhapsody is a commercial model-driven development tool targeted at the de-
velopment of embedded and real-time systems. The Rhapsody modeling environment is
based on the UML. The product has been used in a variety of industrial applications,
implicitly spreading the use of UML [Thales, ThalesOptronics, ECITelecom, Trane,
ZurcherKantolbank, Schwab, MotionControl, ObjectiveControl, Cytyc].

2.3 The UML Meta-model

Modelers use UML to model an abstraction of real world phenomena, e.g. business logic,
prior to implementation. The runtime instances of an implementation and a correspond-
ing UML model reside in two different layers of abstraction as shown in Figure 2.1. The
lowest level, M0, represent the runtime instances (implementation) and the layer above,
M1, represent the UML user-model. The rules by which the UML user-model is de-
fined, i.e. the rules a modeler must obey when using UML for modeling some system,
are defined in the M2 layer [UML1.4.2, p28]. The M2 layer is an abstraction of M1 and
is called the meta-model of UML.. The meaning of meta is data about other data and as
a result the meta-model of UML is the model that describes the UML user-model. The
meta-model prevents modelers from inventing their own constructs or interpretations of
existing constructs at the user-model level. Above the UML meta-model level is the
meta-meta-model (M3) called the Meta-Object Family (MOF), which is the language
used to build meta-models, e.g. the UML meta-model.

Figure 2.1 is described in more detail in the following to allow a deeper understand-
ing of the underpinnings of UML.

The boxes in layers M1-M3 are classes of that layer. The general concept of a class is
applicable to all three layers, but with a twist: a class is a cohesive collection of data that
specifies the structure and behavior of the classes created from the class (its instances).
The subtlety of a class having classes as instances is a feature of meta-modeling. Only
classes in the M1 layer has “real” objects as instances. The classes in layers M1-M3
comprise a model at each layer. The main idea of a model is also applicable to all
three layers. A model is a description designed to show the main features of a concept.
The concept of each layer varies: the concept of a M1 model is to model real-world
phenomena. The concept of a M2 and M3 model is to model a model. In this thesis
a M1 model is denoted a model. The prefix meta is used to distinguish, from a certain
standpoint (i.e. M1), models at different layers. The meaning of meta-model in this
context is data about a M1 model, i.e. a M2 model (UML meta-model). The same
applies for a meta-meta-model, which translates to data about a M2 model, i.e. a M3
model (MOF). Hence, a class is a constituent of a M1 model (UML user-model), a
**CHAPTER 2. UML**

Figure 2.1: Example of the link between different layers of the UML Meta model

A meta-class is a constituent of a M2 model (UML meta-model) and a meta-meta-class is a constituent of a M3 model (MOF).

The dashed arrows in Figure 2.1 are dependencies. A dependency in this context signifies a model element requiring another model element for its specification [UMLSuperstructure2.1.2, p79]. The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier). The text enclosed in angle brackets is the dependency stereotype.

Beginning from the bottom up, the M0 instance *aVideo* is an instance of the M1 class **Video**. UML allows modelers to model instantiated classes. The class **:Video** is a model of an instance of **Video** as indicated by the dependency stereotyped <<snapshot>> going from **:Video** to **Video**. **:Video** is also an instance of the M2 class **Instance**. The M2 classes **Class** and **Instance** are connected by a M1 construct, the an association. The reason it is possible to use a M1 construct in a M2 context is that the UML 2 specification use a combination of three languages to describe the full UML [UMLInfrastructure2.1.2, p34]. One of those languages is a subset of UML\(^3\). The ori-

---

\(^2\)The M2 class **Instance** is in fact the class **InstanceSpecification** [UMLSuperstructure2.1.2, p99] and it represents an entity at a point in time, i.e. a snapshot.

\(^3\)The practice of defining a language by means of a subset of the language being defined yields a semi-circular description. Understanding the UML specification is possible for two reasons: (1) only a small subset of UML constructs are needed to describe its semantics and (2) additional two languages - the
entation and end-name of the association shows that Instance has an attribute named classifier of type Class. According to the UML specification, the attribute classifier is of type Classifier [UMLSuperstructure2.1.2, p99], and the M2 class Class inherits Classifier [UMLSuperstructure2.1.2, p66]. The association shows how Instance can be instantiated as Video. It is possible because it has been modeled with an attribute of type Classifier.

The UML meta-model can be extended through Profiles or first-class extensions handled through MOF. With Profiles, the UML meta-model can be adapted to include (not remove nor alter) constructs not part of UML released by OMG [UMLInfrastructure2.1.2, p189]. It is a generic extension mechanism for customizing the UML meta-classes for particular domains and platforms. Profiles are defined using stereotypes, which define how an existing meta-class, e.g. Class, Property or Operation, may be extended. A Profile is a collection of such stereotypes that collectively customize UML for a particular domain. Profiles suffice if the existing properties of the UML meta-classes makes sense to the domain. If, however, some properties are misplaced or unaligned or if entire meta-classes are missing, first-class extensibility through MOF is an option. First-class extensibility handled through MOF impose no restrictions on what is allowed to do with the UML meta-model, e.g. add or remove meta-classes.

2.4 Choosing versions of UML for comparison

The Rose-VDM++ Link is based on UML 1.1 [UMLMan] and the model transformation developed as part of this thesis is based on UML 2.1.2. Consequently, both specifications of UML must be investigated to uncover differences that may influence the design of the model transformation rules. UML 1.4.2 has been chosen to represent UML 1.x. The rationale for choosing version 1.4.2 is that it is the only UML specification by OMG certified as an international standard by ISO and thus the version recognized by most developers. UML 2.1.2 has been chosen to represent UML 2.x because it is the latest published version of UML by OMG.

UML 1.4.2 is denoted UML 1 and UML 2.1.2 is denoted UML 2.

2.5 UML 1

This section presents the notion of UML 1 Class Diagram (CD) and Sequence Diagram (SD) followed by a description of the model constructs constituting each diagram type. Constructs not applicable to the model transformation between VDM and UML are not described in this section. Interested readers are referred to Appendix B for more information on excluded UML 1 constructs.


2.5.1 Class Diagram

A CD consists of interconnected classes and interfaces. A class is a definition of behavior, structure and relationships shared by multiple instances of the class, denoted objects. A class can be concrete or abstract. A concrete class can be instantiated as opposed to an abstract class which can only be inherited from, even if it contains implementation code. Connections among classes constitute relationships that can take the form of associations or generalizations.

Class

A class in an UML 1 CD is an instance of the meta-class Class. Figure 2.2 depicts a condensed CD from the UML 1 specification and shows the meta-attributes of class Class [UML1.4.2, p44,45,47]. It has the following meta-attributes:

isActive: Specifies whether an instance of a class maintain its own thread of control.

isAbstract: Specifies whether a class can have a direct instance or not.

ownerScope: Specifies whether the attributes or operations (see Figure 2.2) of a class are accessible directly via the class or via an instance of the class. Possibilities are instance or class [UML1.4.2, p106].

feature: Specifies an ordered list of attributes and operations, owned by the class (instances of Attribute and Operation).

Figure 2.2: A condensed CD showing the attributes of meta-classes Class, Attribute and Operation [UML1.4.2, p44,45]
As mentioned above, an Operation is an owned meta-attribute of Class [UML1.4.2, p56], and it too has meta-attributes, as described below.

ownerScope, isAbstract: Have the same semantic meaning as described above for Class.

visibility: Denotes how the class to which it refers is seen outside the enclosing name space. Possibilities are public, protected and private.

isQuery: Specifies whether execution leaves the state of the system unchanged. A value of false indicates that side-effects may occur.

**Parameterized class**

A parameterized class is a generic class with one or more unbound formal parameters. The parameter can be of any type and can be used in the operations of the class. To be a meaningful constituent of a CD, the unbound parameters of a generic class must be bound to some types [UML1.4.2, p54].

![Figure 2.3: An example of a parameterized class](image)

Figure 2.3 shows a generic class Set with one unspecified parameter T. The class EmployeeSet has T bound as type Employee as specified by the \(<\langle bind\rangle\)> dependency to Set.

**Nested Class Declarations**

A class declared within another class belongs to the namespace of the other class and may only be used within it. This construct is primarily used for implementation reasons and for information hiding [UML1.4.2, p234].

**Association**

An association defines a semantic relationship between classes. It consist of two or more ends, each specifying a connected class and may optionally have a name [UML1.4.2, p251]. A binary association connects exactly two classes and a n-ary association may connect several classes. Figure 2.9 shows the relevant meta-classes to understand the definition of an Association.
CHAPTER 2. UML

Figure 2.4: The class NestedClass is declared within the DeclaringClass, hence it is restricted to the namespace of the DeclaringClass [UML1.4.2, p239].

Figure 2.5: Association examples both navigable, composite and ordered.

(a) Showing a navigable association. (b) Showing a composite association with a ordered set.

Figure 2.5a shows how a navigable association is visualized in a diagram. Each Wheel may belong to one Car, and one Car may have multiple wheels. Figure 2.5b shows a Polygon which has a composite association to three or more ordered Point. Class Polygon is the only class with knowledge of class Point, hence instances of Point are constructed and destructed by Polygon alone. Options for enhancing the power of expression of associations are described in the following text.

**Xor association:** Denotes that only one of a set of possible associations may be instantiated [UML1.4.2, p50]. See Figure 2.6 for an example.

Figure 2.6: Xor Association. The dotted line annotated with \{xor\} denotes only one of the two classes may be instantiated by an instance of Account [UML1.4.2, p252]

**N-ary association:** An n-ary association is an association which spans three or more classes, see Figure 2.7. The lines connecting the owner of the association to multiple targets are joint through a diamond-shape. Although an n-ary association is composed of several lines, it is semantically one association. Multiplicity (ex-
2.5. UML 1

explained further below) may be annotated to an n-ary association, but with less rigor compared to a binary association (i.e. between exactly two classes) [UML1.4.2, p259]. Figure 2.7 shows a simple n-ary association with little semantics. It could, however, be further annotated to show the navigability and multiplicity.

![Figure 2.7: An N-ary association. The semantics of the depicted n-ary association is merely that an association of some kind exist among the three classes](image)

Most of the interesting details about an association are attached to its ends. The meta-class AssociationEnd is represented. The following additions can elevate the level of precision to an association-end.

**Multiplicity:** The multiplicity of an association-end specify the number of allowed instances the class located at the opposite end of the association may have knowledge of [UML1.4.2, p256]. A multiplicity is a range of non-negative integers. An example can be seen in Figure 2.5a and 2.5b.

**Ordering:** If the multiplicity is greater than one, the set of related instances may be either ordered or unordered. Default is unordered (duplicates are prohibited) [UML1.4.2, p253]. It is in effect a constraint on the association. An example can be seen in Figure 2.5b.

**Qualifier:** A qualifier is an attribute or list of attributes whose values serve to partition the set of instances associated with an instance across an association [UML1.4.2, p257]. See Figure 2.8 for an example.

**ScopeKind:** Is an enumeration that denotes whether an association belongs to individual instances or an entire classifier. Its values are \{class\} or \{instance\}, respectively.

**Navigability:** When placed on a target end, specifies whether traversal from a source instance to its associated target instances is possible. Specification of each direction across the association is independent. A value of true means that the association can be navigated by the source class and the target role-name can be used in navigation expressions [UML1.4.2, p52].
Figure 2.8: The qualifier accountNo designates that every instance of Person is identified by zero or more qualifier instances. It also shows that every instance of Person may be identified by an infinite number of accountNo instances.

Visibility: Specified by a visibility indicator (+, #, -, or explicit property name such as public) in front of the name of the association. Possibilities are:

- public, where other classes may navigate the association and use the name of it in expressions,
- protected where descendants of the source class may navigate the association and use name of it in expressions, and
- private, where only the source class may navigate the association and use the name of it in expressions.

Figure 2.9: A condensed CD showing the class hierarchy of the meta-class Association.

Generalization

The terms child, subtype, and subclass are synonymous and mean that an instance of a class being a subtype of another class can always be used where an instance of the
latter class is expected. The terms superclass and generalization and their counterparts subclass and specialization, are the preferred terms in this thesis. Generalization is used to classify classes. A class (abstract or concrete) is a superclass if it contains signatures of operations or implementation code common to one or more specializing classes, i.e. subclasses that inherit the superclass. See Figure 2.10 for an example. The superclass is thus a generalization of the subclasses and the subclasses are specializations of the superclass.

Figure 2.10: An example of generalization. A Whiteboard is both a Rectangle and a Shape, i.e. an instance of Whiteboard can be used where the former classifies are expected. Instances of Shape and Rectangle, however, can not be used where Whiteboard is expected.

**Stereotype**

Classes may be annotated with stereotypes. A number of built-in stereotypes exist that can be utilized to denote certain intentions of a stereotyped class, e.g. control or semaphore, and is in effect a new meta-class introduced at modeling time [UML1.4.2, p221] as it is possible to invent new stereotypes (Rose-VDM++ Link uses stereotypes. See section 2.5.3).

**Constraint**

In the UML meta-model, Constraint is used to restrict elements. The restriction can be stated in natural language, or in different kinds of languages with a well defined semantics. The meta-class Constraint has the attributes constrainedElement and body. The first specifies an element to constrain and the second a semantic condition or restriction on that element.

### 2.5.2 Sequence Diagram

A UML 1 SD is made up of the meta-classes Collaboration and Interaction as shown in Figure 2.11. The former is the structural description of the participants, i.e. objects, and Interaction is the description of their communication patterns. The

---

4 Or CollaborationInstanceSet and InteractionInstanceSet, due to the type-instance dichotomy: each element has a dual character, e.g. Class-Object, Association-Link, etc. [UML1.4.2, p206].
structure of the objects that participate in an SD is called a Collaboration. The communication pattern performed by the objects to accomplish a certain task is called an Interaction [UML1.4.2, p125].

Figure 2.11: The meta-classes involved in a SD are shown. [UML1.4.2, p127]

As seen in Figure 2.11, the meta-class Collaboration includes a set of ClassifierRoles and AssociationRoles that defines the objects of the SD. ClassifierRole and AssociationRole defines usage of an instance of Classifier and Association, respectively. That is, ClassifierRole represents the classes and attributes involved in a particular SD [UML1.4.2, p137]. The meta-class Interaction is defined in the context of a Collaboration and it contains a set of partially ordered messages, each specifying one communication, e.g. which operation to be invoked [UML1.4.2, p139].

Each object is called an lifeline and is represented by a box with a vertical dashed line stemming from the lower center of the box. An example is shown in Figure 2.13. The interaction between objects is facilitated by messages between lifelines. The message is drawn as arrows between and is a communication that conveys information, e.g. about an operation to invoke or an instance to create.

An arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction or for identifying concurrent threads of control. An arrow may also be labeled with a condition and/or iteration expression [UML1.4.2, p283].

A connected set of arrows may be enclosed in a separate diagram and marked as an iteration. The continuation condition for the iteration may be specified at the bottom of the iteration. If there is concurrency, some arrows in the diagram may be part of the iteration and others may be single execution. Various labels (such as timing constraints) can be shown either in the margin or near the transitions or activations that they label. Figure 2.12 shows how the lifeline may be split.
into two or more concurrent lifelines to show conditionality or concurrency. The latter may be the case if the conditions are not mutually exclusive. The lifelines may merge together at some subsequent point.

![Sequence Diagram](Figure 2.12: Sequence diagram showing creation of objects along with parallelism.)

Four objects and one actor\(^5\) are shown in Figure 2.12. The collaboration is triggered by a call to \texttt{op()} on \texttt{ob1} and results in numerous calls as time progresses according to an unknown metric. \texttt{ob4} splits its lifeline, suggesting multi-threaded execution. \texttt{ob1} and \texttt{ob2} is destroyed at different points in time and the method-calls \texttt{foo(x)} and \texttt{bar(x)} have conditions to be satisfied prior to execution.

If a diagram loses its clarity due to information overhead (e.g. it exceeds a sheet of paper) it can be split up as seen in Figure 2.13 for an example. In such cases, the cut between the diagrams can be expressed in one of the diagrams with a dangling arrow leaving a lifeline but not arriving at another lifeline, and in the other diagram it is expressed with a dangling arrow arriving at a lifeline from nowhere.

### 2.5.3 Rose-VDM++ Link

This section addresses the extent to which model transformation from VDM to UML is possible using the Rose-VDM++ Link (RVL) tool. RVL allegedly omits VDM and UML.

\(^5\)An intervening entity; a Use Case concept [UML1.4.2, p273].
constructs that can only be described in one of the two languages [UMLMan, p27]. This is not entirely true, however, since RVL does omit UML 1 construct that are expressible in VDM. This section first present UML 1 constructs that RVL supports. Second, the UML 1 constructs omitted by RVL are presented. In this section, UML 1 is referred to simply as UML.

**Rose-VDM++ Link Included UML 1 constructs**

VDM classes are equivalent to UML classes and can be mapped directly to and from VDM.

Values and instance variables are represented as attributes of a class in UML. RVL use the stereotypes <<instance variable>> and <<value>> to distinguish between instance variables and values, respectively. The need for such a distinction exist because values are read-only and instance variables are not [UMLMan, p27].

Operations and functions map into the operations section of the UML class. They are distinguished by the stereotypes <<operation>> and <<function>> because functions are constrained from interaction with values or instances variables. Operations on the other hand are capable of altering the state of an instantiated class by manipulating instances variables [UMLMan, p28].

Operations and functions may be explicit or implicit, i.e. implemented to some degree or not implemented at all. Both types map to the same syntax in UML, hence it is impossible to tell the difference between the two. RVL deals with this issue by saving the implementation of explicit functions or operations and apply the following rules when mapping functions or operation from UML to VDM [UMLMan, p28]:

---

Figure 2.13: Shows how to connect SDs by the use of pointing to and from nowhere [UML1.4.2, 284].
Rose-Link Rule 1: If a function is already defined in VDM, it is mapped into the same kind (implicit or explicit) as defined in VDM.

Rose-Link Rule 2: If a function is not known in VDM (i.e. the function was defined at the UML level) it is mapped as explicit.

Objects (instances of VDM classes) may have relations to other objects through the object reference type [Fitzgerald&05, p78]. These relations correspond to UML associations and are represented by an arrow from the client class towards the class being referenced. The arrow indicates navigability towards the referenced instance. The ends of an association can be given a name which indicates the intended role of the association. Binary associations, i.e. an association among exactly two classes, renders the representation of the referenced object as an attribute in UML superfluous, because it is already given by the name of the association-end. Hence it is represented by the association alone.

In listing 2.1 a VDM model and its corresponding UML 1 Class Diagram generated by RVL is shown.

```vdm
class Person
types
day = nat;

instance variables
name : seq1 of char := "My name";
skills : set of Skill;
favouriteDishes :
  seq of char := [];
appointments : map day
to Appointment;
end Person

class Appointment
end Appointment

class Skill
end Skill

Listing 2.1: VDM model of a Person
```

Fig. 2.14: UML visualization of the VDM model generated by RVL

It is possible to relate one instance to several other instances by using the following VDM constructs:

- **set of objref:** A one-to-many association.
- **seq of objref:** A one-to-many association of ordered elements.
- **seq1 of objref:** The non empty sequence of object references is represented by the range 1..* at the many-end of the association.
[objref]: The optional object reference is represented by the range 0..1 to the association.

Associations resulting from use of seq and seq1 are decorated with the constraint \{ordered\} because sequences are ordered [Fitzgerald&05, p136].

Members of a class may be declared static. A static member can be accessed directly from the class as opposed to a non-static member that requires an instance of the class. RVL distinguishes between the two cases by prepending the character $ to the name of an value and underscores the name of an operation, as shown in listing 2.2 and Figure 2.5.3.

class Owner
values
  nonStaticValue : nat = 0;
  static staticValue : nat = 1;
instance variables
  private nonStaticVar : Owned;
  private static staticVar : Owned;
end Owner

class Owned
end Owned

Listing 2.2: Owner

Mappings in VDM are used to model relationships between the values of one set, referred to as the domain, and the values of another set, referred to as the range [Fitzgerald&05, p167]. A mapping is expressed by the constructs map and inmap. The first denotes the type of all finite mappings between any two types, the mapping operates on, i.e. many-to-one. The latter denotes the injective version of the first, i.e. a one-to-one mapping between two types. The constructs map or inmap result in a qualified association in UML. The qualified association can be assigned a name, which will be the type of the domain of the map. See Figure 2.5.3 for an example of map.

The keywords is subclass responsibility in VDM designates delegation of responsibility, i.e. inheritance relationship, which translates to an generalization arrow in UML. A VDM class containing a delegated operation of function is translated to an abstract class in UML, denoted by an italicized class name.

Rose-VDM++ Link Excluded UML 1 constructs

RVL omits UML 1 constructs that can in fact be represented in VDM. These constructs are presented in this section with a proposition of how to map them to UML 1. Each
description contains a reference to section 4 on static model transformation, where explanations of the transformation rule for each VDM and UML construct can be found.

**Constraints:** Classes, operations or parameters cannot be constrained in RVL. Nevertheless, possibilities exist for specifying simple textual constraints in UML. The meta-class `Constraint` defines such a semantic condition or restriction expressed in text. A number of textual constraints are defined as enumerations in the UML meta-model, e.g. `{Xor}` for use with `Associations`, as showed in Figure 2.6 [UML1.4.2, p50,252]. Such an annotation can be mapped to VDM as an Union type [Fitzgerald&05, p74]. More complex constraints, such as invariant, should not be mapped to UML due to the overhead of textual information it will represent, thus bloating the UML diagram. See section 4.5.

**N-ary association:** An association may be binary or n-ary [UML1.4.2, p259]. The latter is not possible to map from VDM to UML using RVL, although it can be mapped as a product type [UML1.4.2, p75]. See section 4.6.

**Active class:** VDM classes may contain a `thread` section to enable modeling concurrent systems. The UML counterpart is an active class (`isActive=true`). RVL ignores VDM classes with a `thread` section, however, such a class can be mapped as an UML active class. See section 4.9.

**Template class:** A generic class has one or more unbound parameters. Such a construct is not part of VDMTools [Fitzgerald&05, vii]. However, the open source initiative named Overture Project develops a formal modeling method, Overture Modeling Language (OML), founded on VDM. In this context an extension of the VDM formal specification has been made as part of a master's thesis project to include generic classes [Christensen07], hence it is possible to represent generic classes in the Overture variant of VDM. See section 4.12.

**Inner class:** A UML class may have inner classes, i.e. nested classes confined to the name space of the host class. The idea of inner classes is to limit visibility of a class further than possible by use of name spaces. Currently, VDM does not have the concept of inner classes. Extending VDM to support inner classes requires access modifiers on classes, otherwise the concept of inner classes is lost. In listing 2.3 a suggestion to extended the VDM syntax is incorporate to enable the concept of inner classes.
Listing 2.3: Example of extended syntax for nested classes support.

In listing 2.3 a classes **Owner** is shown. The specification in extended by a visibility **public** and a inner class definition **classes** to support declaration of nested classes.

Another approach, that which is taken in this thesis, is to map definitions of VDM data types as UML nested classes. The **types** part of a VDM class definition contains any data types defined in the class. Such types can be considered inner classes; see section 4.3.

### 2.6 UML 2

This section presents the differences between UML 1 and UML 2 in terms of new and deprecated constructs. Section 2.5 introduced a number of UML 1 constructs in order to explain the level of UML 1 support offered by RVL. The aim of this section is to prepare the reader for the sections describing the transformation rules.

#### 2.6.1 Class Diagram

The basic building blocks of a CD have not changed from UML 1 to UML 2. It still consist of classes with relationships among them. Figure 2.16 gives an overview of the meta-classes constituting a UML 2 class diagram. The changes important to this thesis work are presented in the following.

**Deprecated UML 1 meta-classes**

**Association:** Binary associations and attributes now have different notations for the same concept, i.e. an attribute may represent the navigable ends of a binary association.

**ChangeableKind:** The enumeration has been revoked. It comprised the values \{changeable\}, \{frozen\} and \{addOnly\}, which denoted how an association-end was allowed to be modified. It has been replaced the meta-class **Constraint**, which allows textual constraints to be devised at a desired level of detail using natural language or more formal languages, e.g. OCL. [UMLSuperstructure2.1.2, p74].
2.6. UML 2

Figure 2.16: A condensed CD showing the attributes of meta-classes Class, Attribute and Operation [UMLSuperstructure2.1.2, p48]

**Xor-association:** The special kind of association has been replaced by the use of Constraint.

**Nested class symbol:** The nested class symbol shown in Figure 2.4 in section 2.5 have been deprecated. Normally, connections among classes are some kind of Relationship [UML1.4.2, 45] [UMLSuperstructure2.1.2, 148]. The nested class symbol in UML 1, however, are purely notational and has no supporting meta-class. No specialization of Relationship in UML 2 is used to model nested classes. The notation in UML 2 is simply to prepend the name of a nested class with its owning class (see Figure 4.3 in chapter 4 for an example).

### 2.6.2 Sequence Diagram

The basic building block of a UML 2 SD have not changed. It still consist of lifelines connected by messages. However, some of the underlying meta-classes have changed.

An interaction may be part of Sequence Diagrams, Interaction Overview Diagrams, and Communication Diagrams [UMLSuperstructure2.1.2, p473]. Figure 2.17 shows meta-class Interaction and other meta-classes related to it. Notice that every class, except those related to messages, inherit InteractionFragment. The idea in UML 2 SD is that everything is a piece of interaction and that pieces of interaction may be nested within each other to form more complex interactions.

The idea becomes apparent in Figure 2.18, where it is seen that CombinedFragment and InteractionOperand is also subclasses of InteractionFragment [UMLSuperstructure2.1.2, p488,501].
In UML 2 the concept of CombinedFragments is introduced. The only valid option for describing procedural logic in UML 1 Sequence Diagrams is to separate a collection of messages in a separate diagram, e.g. for modeling an iteration with a condition (see section 2.5.2). In worst case the result is a multitude of different diagrams referencing each other. CombinedFragment mitigates this problem. A CombinedFragment is a piece of interaction, i.e. a part of a Sequence Diagram. At the same time it is a Sequence Diagram by itself. CombinedFragments can be nested within each other. They are
used to increase clarity and keep Sequence Diagrams concise [UMLSuperstructure2.1.2, p483].

As seen in Figure 2.18, CombinedFragment has two attributes, interaction-Operator and operand. The first dictate the semantics of the CombinedFragment, i.e. how it should be interpreted by the reader. Depending on the kind of interaction operator zero, one or more operands separated by a dashed line can be used. The following operators have been selected for use in this thesis:

Alternative (alt): Denotes a choice of behavior akin to an if-then-else block. At most one operand must be chosen. It replaces the UML 1 notation of splitting a lifeline into concurrent lifelines to show conditionality. The syntactical way to define guards to test against is by Continuation [UMLSuperstructure2.1.2, p490]. Continuations have semantics only in connection with Alternative fragments.

Loop (loop): Encloses a series of messages which are repeated. The number of iterations is defined by a pair (minint, maxint) of minimum and maximum repetitions or by a boolean expression.
2.7 Tool support for XML Metadata Interchange (XMI)

The XML Metadata Interchange (XMI) is a standard for exchanging meta data information via Extensible Markup Language (XML) it can be used for any meta model that can be expressed in Meta-Object Facility (MOF). XMI is standardized by the Object Management Group (OMG) [OMGUMLHomepage]. XMI is widely used to exchange UML models by UML modeling tools. The modeling of data in XMI is split into two parts on abstract model and one concrete model which is the vision of OMG. The abstract model represents the semantic information which in the case of UML would be e.g. class definitions, it is an instance of arbitrary MOF-based modeling languages such as UML. The concrete model represents visual diagrams such as Sequence diagrams in UML. For diagrams the Diagram Interchange [UMLDI] (XMI[DI]) which is a standard specifying how the actual diagram should be specified.

2.7.1 XMI incompatibilities between UML modeling tools

There are several incompatibilities between different tool vendors implementing XMI for UML. At the Diagram interchange level the standard are almost nonexistent and even between interchange of abstract models there are multiple incompatibilities. Unfortunately this means that the goal of XMI: Enable interchange of e.g. UML model are rarely possible or very limited. Moreover the new XMI 2.1 standard are even less widespread which as a result limit the interchange of models even more.

To give an impression of how this deviations from XMI influence the interchange of models a list of UML modeling tools is provided together with examples of limitations and deviations from the standard. All the tools listed have support for XMI version 2.1 which at the time of writing are the newest standard released by OMG. Common for all the tools are the widely use of the extension tag as shown in listing 2.4 which enables tool vendors to extend the XMI model with tool specific data. This feature is by no means intended to be used as a substitute for the abstract or concrete model.

```
1  <xmi:Extension extender="Enterprise Architect" extenderID="6.5">
2  </xmi:Extension>
```

Listing 2.4: Example of XMI extension tag.

**Enterprise Architect (EA) [EA71]:** They claim to have full support for UML 2.1.2 and XMI 2.1. This is partly true, but they do not use the XMI[DI]. All diagrams are placed inside a EA extension tag with all data serialized to EA specific elements. In addition to this they placed the information about navigable associations (see figure 2.5a) and placement of qualifiers of associations (see figure 2.8) etc. in extension tags as well. The above information in specified in the abstract model at export but ignored by EA itself at import.
Visual Paradigm for UML (VP-UML) [VP-UML]: They claim to have full support for UML 2.1.2 and XMI 2.1. This is not correct, but compared to EA they use the XMI[D1] to some extent. All diagrams are facilitated by extension tags. The tool is limited in the abstract model e.g. it does not support n-ary associations instead it creates binary associations between a class acting as the diamond of the n-ary which do not exist. This diamond class is represented in the abstract model as a class with no name and an extension.

```
<ownedMember name=""
    xmi:type="uml:Class"
    ...
    <xmi:Extension xmi:Extender="Visual Paradigm for UML">
        <modelType value="NARY"/>
        <nary/>
    </xmi:Extension>
</ownedMember>
```

Listing 2.5: VP-UML n-ary association class

Eclipse UML2 Tools [UML2Tools], Topcased [TOPCASED-UML2], rCos [rCOS]:
Supports UML 2 and XMI 2.1 but have their own name space for UML which reduce the interchange of models. Besides the limitation of n-ary associations like VP-UML and the fact that the UML type are missing from a subpart of elements in the abstract model, it has a good support. Both Topcased and rCos are build on top of Eclipse UML2.

2.7.2 Limitation of Sequence Diagrams in XMI

The XMI presentation of a Sequence Diagram (SD) is limited since there is no way a Message can be linked to a CombinedFragment in the abstract model of the sequence diagrams (see section 2.6.2). This missing link means that a connection between a message of SD and the InteractionOperand in which it may exist are missing from the abstract model, the link only exist in the diagram where most tool vendors use their own standard.

Limitation: From a transformations point of view the use of CombinedFragment in SD is restricted to none, since they do not contribute with any value if no Messages can be related to them.

To solve the problem without breaking the XMI standard a extension tag is inserted in the operand element representing a interaction operand. The new extension tag is provided with a covered element containing the id of the MessageOccurrence-Specification (MOS) at the message sendEvent and receiveEvent to link the message and operand together. This makes it possible through the abstract model to obtain enough information to reproduce the SD by taking the new extension and the
ordering of messages into account. Instead of placing this link in an extension of the operand it should be included in the operand element but it will require the standard to be changed.

```
<ownedBehavior xml:id="idInteraction"
    name="SDLInteraction">
    <fragment xml:id="VDM.11" name="CF1"
        covered="VDM.5 VDM.12" interactionOperator="loop">
        <operand xml:id="VDM.21">
            <Extention extender="umltrans">
                <covered>VDM.14 VDM.6</covered>
            </Extention>
            <guard xml:id="VDM.22" constrainedElement="VDM.21">...</guard>
        </operand>
    </fragment>
    <message xml:id="VDM.20" name="" receiveEvent="VDM.14" sendEvent="VDM.6"/>
</ownedBehavior>
```

Listing 2.6: Setting multiplicity of properties

In listing 2.6 an example of the extension is shown in line 5-10. Where the message at line 12 is linked to the operand.
Chapter 3

VDM

The Vienna Development Method (VDM) is a method comprising a collection of techniques for the formal specification and development of software systems. It is based on the VDM Specification Language (VDM-SL), which is a model-oriented formal specification language. VDM-SL has an extended form, VDM++, which supports modeling of object-oriented systems (denoted VDM from this point). VDM is one of the oldest formal methods, and have been applied to a wide range of industrial projects [Fitzgerald&08a, Fitzgerald&08b].

3.1 History of VDM

In the period of 1964-69, IBM used the meta-language Vienna Development Language (VDL) to define the semantics of PL/1 (Programming Language One) [DinesBjornerPP, Plat&92, Fitzgerald&08b]. The descendant of VDL, Meta-IV, was used by IBM in 1973-75 to develop a PL/1 compiler [DinesBjornerPP]. The development approach taken by IBM became known as the Vienna Development Method, because it comprised several techniques, including, but not limited to, Meta-IV. During the 1970s, different specializations of Meta-IV began to emerge as a result of different application areas. As a consequence, the British Standards Institution (BSI) submitted a standardization proposal to ISO aiming at reconciling the strands into a unity. VDM-SL was ISO standardized in 1996 [ISOVDM96] and the standardization of Meta-IV was called VDM-SL. Extensions incorporating object oriented structuring and handling of concurrency were developed in 1992-94 by the Afrodite project [Fitzgerald&05, p10]. The extended VDM-SL together with improved tool support is collectively denoted VDM++ [Fitzgerald&05, p10]. Further extensions have been made to VDM, most noticeably VDM In Constrained Environment (VICE) which support the modeling of distributed and real-time systems [VDMLangManVICE]. Available tool support for VDM include:

---

1Afrodite has been sponsored by the European Union under the ESPRIT programme (EP6500).
**VDMTools**: Constitutes the leading commercial tools for VDM-SL and VDM++. It is an industry-strength tool set owned, marketed, maintained and developed by CSK Systems, building on earlier versions developed by the Danish Company IFAD A/S [VDMFromWikipedia, IFAD].

**Overture**: An open-source project [OvertureTool] aiming at providing free tool support for VDM++ on top of the Eclipse platform and to develop a framework for interoperable tools that will be useful for industrial application, research and education [VDMFromWikipedia, OvertureTool]. Focus is also on making it easier to use VDM to test new ideas and concepts.

### 3.2 VDM Usage

The purpose of VDM is basically the same as that of UML, i.e. to let developers focus on the critical parts of a software system by abstracting parts of the system away. It is, like UML, designed to be independent of methodologies and programming languages. The great difference between the two is that VDM is a model-based formal method, while UML is visual semi-formal method. If a model is created in UML it is recommended to use a verification tool to ensure a certain degree of completeness and correctness.

VDMTools can be used to validate a VDM model very easy. VDM requires that all constructs to be precisely specified and as a reward for the VDM language supports type check of the model, the ability to run the hole model and assistance of proof construction.

The useful feature that enables code generation exists in many modeling tools for UML, the same is the case for VDM, where the model can be executed at the specification level and the final source that can be generated from the specification.

#### 3.2.1 Banknote Processing

The SIC2000 project at the GAO\(^2\) involved the development of a complex collection of mutually suspicious cooperating software components in a banknote processing system. The goal was to expedite the integration of sensor software and hardware into existing banknote processing systems. This was the first large project at the GAO in which formal methods played a central role. The analysis, design, and test phases of the development were supported by the IFAD VDM-SL Toolbox.

The use of VDM in the SIC2000 project can be considered lightweight in the sense that no formal refinement or proof has been performed. Instead, in the course of the project, the VDM technology became the focal point of the entire development process, providing a unified treatment of analysis, design, documentation, and testing. The development of the formal specification of the SIC2000 project was performed in 1 man-year. The implementation (in the SIC) of the specification was completed in 3 man-months.

\(^2\)Sensor Integration Controller (SIC) project, which was undertaken at the GAO (Gesellschaft für Organisation und Automation).
Modular testing of the SIC was finished in 4 man-weeks: several errors were detected, all but one attributable to an imperfect translation of the specification into code.

The developers of the project believes, that the primary advantage of the VDM technology is the support it provides for the construction of precise and realistic software system models. For complex industrial projects, this is a capability whose value can hardly be overestimated. The price that must be paid for the capability of formulating design ideas in a formal notation, following the implications of design features to their logical conclusions, exposing and codifying all design assumptions in a collection of formal invariants is not excessively high, considering the alternatives.

If VDM had not been utilized in the project, the result would have been a different, a weaker, and eventually a more expensive final product than was produced by using lightweight formal methods [Smith&99].

3.2.2 VDMTools

Most components of the VDMTools tool suite are themselves developed using VDM. This development has been made at IFAD in Denmark and CSK in Japan [Fitzgerald&08a].

3.2.3 ISEPUMS

In a project from the space systems domain, VDM was used to develop a sub-system of the ground segment, which handled processing of messages communicated to the SPOT4 satellite, an earth observation satellite designed by CNES (French National Space Centre) and in service since March 1998. The aim of the project was to apply a knowledge acquisition method in combination with a specification of a real-size application. The combination of a knowledge acquisition method and the use of VDM, with prototyping and multi-modules capabilities, enabled modeling the complicated system and build a first complete executable specification in a reasonable time (5 months). The direct use of the VDM method to build a model immediately revealed the complexity and level of details of the ground segment, leading to frequent meetings with the domain expert to clear up the ambiguous points. Furthermore, VDM helped to find the right choice in terms of abstractions, which results in a smaller system than the one obtained using classical techniques [Puccetti&99].

3.2.4 A Mission Critical Data Handling Subsystem

The data handling subsystem developed using VDM-SL technology is an input- and output service of a large data mining application. On the input side of the data handling subsystem, the arriving messages can be very complex, due to the richness of the message syntax and semantics. This leaves the opportunity for highly ambiguous messages to be send to the system’s, analogous to natural language interpretation. Furthermore, the

---

3The identification and categorization of relevant domain knowledge.
format of the input messages evolves over time (and in practice, so will the data model of the data mining application), so the requirements with respect to the maintainability of the data handling subsystem were very high. Due to the clear need for a very expressive method for specification of the subsystem, VDM-SL was selected (and the IFAD VDM-SL toolkit) as means to implement the data handling subsystem. The system as a whole was developed under a fixed-price, fixed-date contract. About 4800 man-hours (which corresponds to about 10 per cent of the total available project resources) where spent on the development of the data handling component. From the start of the project, the data handling subsystem has been in the critical path of the project plan, mainly due to the sequential nature of the activities. Nevertheless, it was the first subproject to deliver results, on time, within the allocated budget and was accepted by the client without a single change. The application of VDM-SL was a major success and the client has continued to work on the data handling subsystem using VDM-SL and the IFAD tools [Berg&99b].

3.3 Tool support

To aide development in VDM a commercial tool called VDMTools exists, which provides a wide range of features like: extensive static semantics checking, automatic code generation, round-trip mapping to UML 1 class diagram, documentation support, test coverage analysis and debugging support. The tool is built on top of the IFAD VDM Toolbox and currently maintained and developed by CSK. The tool has a comprehensive code generator both to c++ and Java it supports up to 95 per ce of the VDM language [Fitzgerald&08a]. The integrity checker points out all places where potential runtime error could happen named proof obligation e.g. check that all operators are applied correctly. VDMTools enables a model to be split up into multiple files enable concurrent model development additional to this is includes a pretty printer that can format the model and used together with the interpreter for debugging and running the model it can color the model according to the part executed and collect coverage information. In the interpreter it has the ability to check invariants, pre and post conditions when execution the model. A VICE extension to VDMTools exist which enable future modeling of time and deployment of resources in a distributed architecture. A external plug-in for VICE exists to enable a graphical representation of the execution. The VDMTools has been used in many industrial projects: ConGorm, Dust-Expert, The development of VDM-Tools, TradeOne, Sony/FeliCa Networks etc. [Fitzgerald&08a]. The tool is available on multiple platforms, e.g. Windows, Linux and Mac. Another open source tool available is VDMJ, now part of Overture, which is currently being developed. It includes an parser, type check and an interpreter for debugging. VDMJ is a Java implementation of a subset of VDMTools and supports VDM-SL and VDM++.
3.4 VDM Classes

VDM models are structured into classes familiar from the object oriented world. A class is made up of different blocks that can be arranged arbitrarily and may be empty. Here follows a short introduction to the content of each block.

**Value:** Constant values. Instead of using a constant value directly in the model, a value can be defined. This enhances readability and makes changes effortless.

**Types:** There are two kinds of types, *basic types* and *constructed types* [Fitzgerald&05, p71]. The basic types refer to primitive data types such as integer, boolean, char, etc. as found in conventional programming languages. The constructed types, e.g. union or tuple, are made from primitive types using a *type constructor*.

**Instance variables:** Variables constitute the state of an instantiated class and are themselves instances of a type.

**Functions:** A function takes input parameters and produces a result, with no reference to the instance variables of the object.

**Operations:** Same as functions, but an operation can modify instance variables and are thus able to cause side-effects in the model.

**Thread:** An independent thread of execution for each instance of the class.

**Sync:** Access to shared data among parallel threads are synchronized using permission predicates and mutex constraints.

Inheritance in a class hierarchy is denoted by the keyword *is subclass of* after the name of the subclass.

Name conflicts occur if several class members with identical names are defined in a class. This can be resolved by prepending the respective class name to each class member.

Access modifiers govern the visibility of member declarations within a class and the scope of a class. Member declarations can be declared private, protected or public.

**Private:** The member is only reachable from within the class.

**Protected:** The member is reachable from subclasses in its class hierarchy.

**Public:** The member is reachable from all classes in the model.
### 3.5 Types

A type definition is used to define data types that can be used elsewhere in the model. It consists of a type name and the definition of the type.

An invariant is a boolean predicate on a data type restricting the values in a data type by means of an invariant. Thus, by means of a predicate the acceptable values of the defined type are limited to those where this expression is true [VDMLangMan, p32].

VDM provides the ability to structure data, such as records, collections and sequences. VDM uses type constructors to enable creation of new types from the basic types provided. Table 3.1 shows the basic types available in VDM.

As mentioned above, types representing structured values and collections are build from the basic types using a type constructor. The available constructors are described below.

#### Union types:

A union type can be formed by several component types. The formed union type will contain all values from each of the components.

```plaintext
let address : seq of char * nat * PostalTown = mk_("Street",5,<Aarhus>) in skip;
```

Listing 3.2: Example of a Product type. An address composed of three fields.

### Table 3.1: Basic types supported in VDM

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>true, false</td>
</tr>
<tr>
<td>nat1</td>
<td>1, 2, 3, ...</td>
</tr>
<tr>
<td>nat</td>
<td>0, 1, 2, ...</td>
</tr>
<tr>
<td>int</td>
<td>..., -2, -1, 0, 1, ...</td>
</tr>
<tr>
<td>rat</td>
<td>..., -1/7, -1/356, ..., 1/3, ...</td>
</tr>
<tr>
<td>real</td>
<td>..., -12.78353, ..., 0, ..., 1322.2324, ...</td>
</tr>
<tr>
<td>char</td>
<td>'a', 'b', 'c', ..., '+' , '-' , ...</td>
</tr>
<tr>
<td>quote</td>
<td>&lt; RED &gt;, &lt; CAR &gt;, &lt; QuoteLit &gt;, ...</td>
</tr>
<tr>
<td>token</td>
<td>mk_token(…)</td>
</tr>
</tbody>
</table>

Listing 3.1: Example of a Union type.

Listing 3.1 shows a NewUnionType having the ability to reference a nat or a bool value.

#### Product types:

The product type brings all values from the composed types together in composite structures called tuples, which is different from the union type where all the values are combined. A product type consists of combined types:

```plaintext
let address : seq of char * nat * PostalTown = mk_("Street",5,<Aarhus>) in skip;
```

Listing 3.2: Example of Product type. An address composed of three fields.
Listing 3.2 shows a product type, `address`. It consists of a `street`, `number` and a `town`. To create a product type the make constructor, `mk`, is used.

**Record types:** A Record type is like a product type but with the ability of naming the different fields in the record as shown in Listing 3.3.

```plaintext
Address ::
    street: seq of char
    number: nat
    town : PostalTown

let street = mk_Address("Street",5, <Aarhus>).street in ...
```

Listing 3.3: Let expression creating an instance of the record using the `mk` command. Example of Record type. An `Address` composed of three types and a `let` expression creating an instance of the record using the `mk` command.

**Optional types:** The optional type constructor takes any type as an argument and adds the special value `nil` to it. For example, the type `seq of char` may be extended with the special value `nil`, as shown in Listing 3.4.

```plaintext
Name : [seq of char]
```

Listing 3.4: Example of a optional type.

**Object References types:** Serves as a reference to objects of given class. Listing 3.5 shows a VDM a class `B` with an instance variable `a` whose value is an instance of class `A`. Hereby the instance variable `a` references an instance of the class `A`.

```plaintext
class B
    instance variables
    a : A := new A();
end B
```

Listing 3.5: Example of the creation of an object.

### 3.6 Test Trace

Test traces is a feature for expressing test-cases in a compact form for exhaustive testing of a model. By the use of repeat-patterns on traces, it is possible to express that sequences of operation-calls should be tested in all possible combinations. Since repeat-patterns may be nested within each other, the risk for combinatorial explosion exists, i.e. a trace may result in an excessive amount of test-cases. A test case resulting in an error is caught and if the erroneous test case appears as a subpart of another test case, the remaining execution of such a trace is cancelled. Traces are defined in the `traces` block.
of a VDM class. Each trace must have a name, but a single trace may define several execution paths separated by a semicolon in a sequential order.

```
traces definitions = 'traces', { named trace } ;

named trace = identifier, {'/', identifier }, ':', trace definition list ;

trace definition list = trace definition term, ';' , trace definition term } ;

trace definition term = trace definition
                           | trace definition term, '|' , trace definition ;

trace definition = trace core definition
                  | trace bindings, trace core definition
                  | trace core definition, trace repeat pattern
                  | trace bindings, trace core definition, trace repeat pattern ;

trace core definition = trace apply expression
                      | trace bracketed expression ;

trace apply expression = identifier, '.', identifier, '(' , expression list , ')' ;

trace repeat pattern = '*' | '+' | '?'
                      | '{', numeric literal, '}'
                      | '{', numeric literal, '}' numeric literal, '}' ;

trace bracketed expression = '(' , trace definition list , ')' ;

trace bindings = trace binding, { trace binding } ;

trace binding = 'let', local definitions, {' ', local definition }, 'in'
               | 'let', bind, 'in'
               | 'let', bind, 'be', 'st', expression, 'in' ;
```

The traces syntax shown above is represented as an Abstract Syntax Tree (AST) to make it available for computational use. The following descriptions present each traces construct in the context of an AST, e.g. `named trace` which consist of at least one `identifier` followed by a `trace definition list`, is represented as the `NamedTrace` in the AST. It is seen, that `NamedTrace` defines a name, `name`, and a trace definition, `defs`, as a `seq of char` and `TraceDefinition`, respectively, which corresponds to `identifier` and `trace definition list` in the traces syntax. Hence, the AST is merely a way to represent language syntax so that a computer can process it.
3.6. TEST TRACE

**Named trace:** A named trace consists of a name and a sequence of trace definitions, see Listing 3.6.

```plaintext
NamedTrace ::
  name : seq of char
  defs : TraceDefinition;
```

Listing 3.6: NamedTrace.

**Trace definition:** A trace definition consists of a sequence of `let` or `let be` expressions, a test which can be a `TraceMethodApply` expression of a bracket expression and a optional repeat pattern.

```plaintext
TraceDefinitionItem ::
  bind : seq of TraceBinding
  test : TraceCoreDefinition
  regexpr : [TraceRepeatPattern];
```

Listing 3.7: TraceDefinitionItem.

**Method apply expression:** The method apply expression consists of a variable name which is the name of an instance of a class whose operation is execute upon and lastly a sequence of parameters see listing 3.8.

```plaintext
TraceMethodApply ::
  variable_name : Identifier
  method_name : Identifier
  args : seq of Expression;
```

Listing 3.8: TraceMethodApply.

**Repeat pattern:**

- **Zero of more** `[a*]`: Repeat expression a from 0 to a pre-defined maximum value. If the maximum value is 3, then a would have the trace nil, a, aa, aaa.

- **One of more** `[a+]`: Repeat expression a from 1 to a pre-defined maximum value. If the maximum value is 3, then a would have the trace a, a a, a a a.

- **Zero of one** `[a?]`: Repeat expression a from 0 to 1. The trace of a would then be nil, a.
Exactly \( n \) times \([an]\): Repeat expression \( a \) exactly \( n \) times. If \( n \) is equal to 4, then the trace of \( a \) would be \( a \ a \ a \ a \).

From \( n \) to \( m \) times \([an,m]\): Repeat expression \( a \) between \( n \) and \( m \) times. If \( n \) is equal to 2 and \( m \) is equal to 4, the trace of \( a \) would be \( a \ a \), \( a \ a \ a \), \( a \ a \ a \ a \).

### 3.7 Trace Example

A example of some of the possible trace statements is shown in listing 3.9 from line 9 in the class named `UseStack`. The trace statement at line 12 expanded in listing 3.10 to show how the interpreter in VDM Tools would execute the `trace` statement.

```plaintext
1 class Stack
2 operations
3 public Push3 : nat => ()
4 Push3(e) == ...
5 public Pop : () => nat
6 Pop() == ...
7 end Stack
8
9 class UseStack
10 instance variables
11 s : Stack := new Stack();
12 traces
13 trace1 : s.Push3(1)*
14 trace2 : s.Push3(1)+
15 trace3 : s.Push3(1)?
16 trace4 : s.Push3(1){2}
17 trace5 : s.Push3(1){0,4}; s.Pop()
18 end UseStack
```

Listing 3.9: Example of the definition of traces in a class.

```plaintext
s.Pop()
s.Push3(1); s.Pop()
s.Push3(1); s.Push3(1); s.Pop()
s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()
s.Push3(1); s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()
```

Listing 3.10: Trace statement expanded from listing 3.9 line 17.
Chapter 4

Static Model Transformation

This chapter describes the transformation rules for each VDM construct chosen to be part of the static model transformation. The rules are defined in order to enable a transformation between VDM and UML. Each rule use a VDM construct as a base to describe how the rule successfully transform the VDM construct to UML.

Not all VDM constructs have a concrete representation in UML, hence some VDM constructs are omitted from the model transformation, e.g. invariants and pre-conditions. The reason is the different intentions of VDM and UML: UML is a well-defined visual language, thus the structure and functionality of a system is expressed mainly by means of diagrams with a degree of rigour [UMLInfrastructure2.1.2, p33]. VDM, on the contrary, has a mathematical semantics for proving properties about a model [Fitzgerald&05, p4]. The detailed syntactical statements of a VDM model has little purpose in a visual representation and will only serve to bloat a UML diagram. The transformation rules for a static transformation describes how VDM class constructs are related to certain UML meta-classes comprising UML diagrams.

4.1 Classes

VDM classes have a one-to-one relationship to UML classes.

<table>
<thead>
<tr>
<th>Transformation Rule 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDM classes are mapped as the UML meta-class <code>Class</code></td>
</tr>
</tbody>
</table>

4.2 Visibility

The UML meta-class `VisibilityKind` is an enumeration of the different visibilities an element can have. Of those elements, `package` is left out because it does not have a VDM counterpart.
CHAPTER 4. STATIC MODEL TRANSFORMATION

Transformation Rule 2
The visibility of VDM instance variables, values, functions and operations are mapped as a *subset* of the UML enumeration VisibilityKind comprising *public*, *private* and *protected*.

The VDM static keyword allowing access to classes, values, instance variables, functions or operations without having a specific instance, is mapped as the isStatic property of the UML meta-classes Class, Property or Operation respectively.

Transformation Rule 3
VDM static is mapped as the isStatic property of the UML meta-class Class, Property or Operation respectively.

4.3 Data types
Data types may be defined in a types block of a VDM class. The definition of a data type within a VDM class resembles the concept of a nested class in UML, except that instances of data types are identified only by their value in contrast to classes, which may also be identified by reference. See section 5.1.3 for the rationale behind the decision.

Transformation Rule 4
Data type definitions are mapped as the UML meta-class Class and are referenced, and thus nested, through the meta-attribute nestedClassifier of the owning class. Notice that this rule is not specified or implemented, hence Figure 4.3 is not generated by the tool made as part of this thesis.

Listing 4.1 shows a VDM class Car with a data type Manufacturer. Figure 4.3 shows how the instance variable mfacturer of type Manufacturer is mapped as the association between Car and Car::Manufacturer with the role name mfacturer. The figure also shows how the data type Manufacturer is mapped as the class Car::Manufacturer, i.e. as a nested class of class Car.
4.4 Instance variables and values

Instance variables and values are defined in the block instance variables and values, respectively, of a VDM class. They are the attributes (i.e., the properties) of a VDM class which hold the state of an object. In UML, the attributes of class may be represented as associations or as attributes.

Transformation Rule 5

Instance variable and value definitions are mapped as the UML meta-class Association, if:

5 a: The type is an object reference type, or
5 b: The type is not a basic data type [Fitzgerald&05, p64,71].

Listing 4.2 shows VDM classes Order and Customer. The instance variable customer is of type Customer and is thus identified from a class, i.e. its type is an object reference. Figure 4.4 shows the corresponding UML class diagram for Order and Customer. The association between Order and Customer is the instance variable customer (rule 5 a). The class Order::OrderIdType is the data type OrderIdType prefixed Order because it is a nested class. The association between Order::OrderIdType and Order is the value id, as indicated by the name and multiplicity at the Order::OrderIdType end of the association (rule 5 b).

Listing 4.2: VDM classes with instance variables showing a UML association.
CHAPTER 4. STATIC MODEL TRANSFORMATION

Transformation Rule 6
Instance variable and value definitions are mapped as the UML meta-class Property [UMLSuperstructure2.1.2, p48,p139], if the type is a basic data type [Fitzgerald&05, p71]. Instance variables and values are distinguished by the meta-attribute isReadOnly. Notice: rules 9 and 12 is an exception to this rule.

<table>
<thead>
<tr>
<th>VDM concept</th>
<th>Property::isReadOnly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance variables</td>
<td>false</td>
</tr>
<tr>
<td>Values</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 4.1: The meta-attribute isReadOnly distinguishes instance variables and values

Transformation Rule 7
The initial value of instance variables and values definitions are mapped as the property default of the UML meta-class Property.

Transformation Rule 8
The VDM optional type is mapped as the UML meta-class MultiplicityElement with the properties lower = 0 and upper = 1.

Listing 4.3 shows a VDM class Address which have a value zip and an instance variable houseNumber, both of the basic type nat. Both zip and houseNumber are mapped as attributes of the corresponding UML class Address, as shown in Figure 4.4, because they are a basic type.

```
1 class Address
2 values
3   zip : int = 8000;
4 instance variables
5   houseNumber : nat;
6 end Address
```

Listing 4.3: The VDM class with a value and an instance variable showed as UML attributes.

Figure 4.3: UML attributes.
4.5 Union Types

A union type is a union of values from different types [Fitzgerald&05, p74]. Listing 4.4 shows an example of a Person that may be either male, female or bool\(^1\), as indicated by the union gender.

Transformation Rule 9

A union type is mapped as the meta-class Association between the owning class and the types specified in the union type. The resulting associations are decorated with a textual constraint \{xor\}. The constraint is an instance of the meta-class Constraint. Notice, that if a member of a union type is a basic type, it is mapped as a separate UML class. This is an exception to rule 6.

Listing 4.4 shows a VDM classes Person, Female and Male. The instance variable gender is mapped as three associations each connecting Person to one of the possibilities of gender. The textual constraint \{xor\} spanning all associations will allow only one possibility to be chosen.

```vdm
1 class Person
2  instance variables
3     name : seq of char;
4     age : nat;
5     gender : Male | Female | bool;
6  end Person
7 class Male
8 end Male
9 class Female
10 end Female
```

Figure 4.4: A textual constraint \{xor\} spanning three associations.

4.6 Product Types

A product type is a composite structure, which consists of tuples of values. Listing 4.5 shows three uses of product type and Figure 4.5 shows the UML counterpart: A product type declared as a data type will result in subfigure 4.5a. A product type declaration with more expressive power when visualized in UML is the VDM instance variable addressWork and addressBook of Listing 4.5, which result in subfigures 4.5b and 4.5c: a n-ary Association which makes it possible to have multiple participants in an association. Note that part set of Name in line 8 of Listing 4.5 is shown as the multiplicity of 0..* in Figure 4.5c.

\(^1\)For the sake of argument, a Person may be regarded false if the gender is indefinable.
class Person
types
    Address = City × County × Street;
instance variables
    addressHome : Address;
    addressWork : City × County × Street;
    addressBook : City × County × set of Name × Street;
    age : nat;
    name : Name;
end Person

Listing 4.5: Example of a Person where a product type are used to model the addressWork and addressBook. The addressHome is declared from a explicit type.

Transformation Rule 10
A VDM product type maps to:

10 a: The UML meta-class Class if it is declared as a data type. See figure 4.5a.

10 b: The UML meta-class Association if it is not defined as a type (i.e. it is anonymous). See figure 4.5b and 4.5c.

Each association-end that represents an entry in the product type is named according to the product type. The types constituting the product type are sorted alphabetically according to the name of the types used in the product type.

4.7 Collections

VDM define three constructs to model collections:

set: Repetition and order of elements are insignificant, i.e. multiple copies of an element and the order of elements are disregarded.

seq: Repetition and order of elements are significant, i.e. multiple copies of an element are distinguishable by the order in which they appear.

seq1: Equal to seq, except that an empty sequence is illegal.

Listing 4.6 shows four VDM classes. Class Order models two sequences, contacts and subOrders, of which the former must contain at least one element. Class SubOrder models a set of Product. The two sequences, contacts and subOrders, are mapped as associations between Order and Contact and Order and SubOrder, respectively,
4.7. COLLECTIONS

(a) VDM product type where the data type is explicitly declared.

(b) VDM product type where the type is implicit.

(c) VDM product type represented in UML where the type is both implicit (addressBook) and explicit (Name), stated

Figure 4.5: UML representation of constructs from Listing 4.5
as shown in Figure 4.7. The constraint \{ordered\} on the associations indicate that the type is \texttt{seq} and the different multiplicities further differentiates the types as \texttt{seq} and \texttt{seq1}. The \texttt{set products} are mapped as an association between classes \texttt{SubOrder} and \texttt{Product}.

```plaintext
class Order
  instance variables
  contacts : seq1 of Contact;
  subOrders : seq of SubOrder;
end Order

class Contact
end Contact

class SubOrder
  instance variables
  products : set of Product;
end SubOrder

class Product
end Product
```

Listing 4.6: VDM classes with collections.

Transformation Rule 11
The VDM constructs \texttt{set}, \texttt{seq} and \texttt{seq1} is mapped as the UML meta-class \texttt{Association} which may be decorated with a textual constraint defined by the meta-attribute \texttt{isOrdered} in addition to a multiplicity at both ends. Table 11 shows how the above-mentioned VDM constructs are mapped.

<table>
<thead>
<tr>
<th>VDM construct</th>
<th>Ordered</th>
<th>Target Multiplicity</th>
<th>IsUnique</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{set}</td>
<td>false</td>
<td>lower=0, upper=*</td>
<td>true</td>
</tr>
<tr>
<td>\texttt{seq}</td>
<td>true</td>
<td>lower=0, upper=*</td>
<td>false</td>
</tr>
<tr>
<td>\texttt{seq1}</td>
<td>true</td>
<td>lower=1, upper=*</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 4.2: Transformation rules for VDM constructs modeling collections

\footnote{The meta-class \texttt{Association} has two attributes of type \texttt{Property} which are the end of an association. Those ends may be ordered, indicated by meta-attribute \texttt{Property::isOrdered}.}
4.8 Relationships

The VDM constructs `map` and `inmap` are used to model unique relationships between values of one set, the domain, and another set, the range. The order of elements in the domain and range are insignificant.

`map`: Denotes the type of all finite mappings between values of the domain and range. The relationship between values of the domain and range is many-to-one, i.e. one or more values of the domain map to exactly one value of the range.

`inmap`: Denotes the injective version of the first, i.e. a one-to-one relationship one value from the domain map uniquely to one value in the range.

Listing 4.7: VDM class with a map showing a UML qualified association.

Listing 4.7 shows a VDM class `Order` with three mappings. Figure 4.8 shows that all ranges are mapped as separate classes, regardless of their type. In the case of `charMap`, this is an exception to rule 6. Observe that `nat` is mapped as `int`. 
 CHAPTER 4. STATIC MODEL TRANSFORMATION

CHAPTER 4. STATIC MODEL TRANSFORMATION

62

Transformation Rule 12
The VDM constructs map and inmap are mapped as the UML meta-class Association with a qualifier. The domain is specified by the qualifier, which is located at the source class. The range is specified by the target class. Notice, that if the range is specified by a basic type it is mapped as a separate class. This is an exception to rule 6.

<table>
<thead>
<tr>
<th>VDM construct</th>
<th>Qualifier end</th>
<th>Target class end</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>inmap</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 4.3: Transformation rules for VDM constructs modeling relationships between two sets. See figure 4.8

4.9 Thread
A VDM class with a thread block has an independent thread of control. It corresponds to an active class in UML.

Listing 4.8: A VDM class with a thread definition.

```vdm
1 class SensorController
2 thread
3 while true do
4 skip;
5 end SensorController
```

Figure 4.9: A UML Active class.
4.10 Generalization

A VDM class may inherit several classes using the keyword *is subclass of*. It is possible to model generalization in UML, including the notion of multiple inheritance.

Listing 4.9 shows a VDM class `AmphibiousVehicle` that inherits two other classes, `Vehicle` and `MaritimeVessel`. The three classes are mapped as UML classes with generalization arrows indicating the inheritance relationship between `AmphibiousVehicle` and the two other base classes.

Transformation Rule 14

A VDM class with the keyword *is subclass of* followed by class-names is mapped as the UML meta-class `Generalization`, with the attributes `general` and `specific` referencing the superclass and subclass, respectively. More than one subclass results in more than one instance of `Generalization`.

4.11 Abstract

A VDM class may delegate function or operation definitions to subclasses, in which case the class is abstract. An UML abstract class is distinguished from a conventional class by an italicized name.
Listing 4.10: VDM class with an operation that is subclass responsibility.

Listing 4.10 shows a VDM class `AbstractSensor` with an operation that delegates its responsibility to a subclass. `AbstractSensor` is mapped as an UML abstract class, as shown in Figure 4.11 by the italicized name of the class.

<table>
<thead>
<tr>
<th>Transformation Rule 15</th>
</tr>
</thead>
</table>
|A VDM class with the keyword `is subclass responsibility` as a function or operation body is mapped as the UML meta-class `Class` with the meta-attribute `isAbstract` set to `true`.

4.12 Generic classes

VDM generic classes has one or more unbound parameters. Currently this is only defined in the OML AST used in the Overture parser. There is not a complete integration of VDM generic parameters and thereby no type check or interpreter functionality available to support this feature at the moment.

Listing 4.11: A VDM class with template parameters. Only supported in Overture.

Listing 4.11 shows a VDM class `List` with two template parameters `T1` and `T2` («[T1, T2]>). The class `List` is mapped as a UML class with the two template pa-
4.13 Operations and Functions

A VDM function or operation can be transformed into the UML meta-class Operation. A function does not change the internal state of the owning class, as opposed to an operation which may change the internal state of the owning class. Both functions and operations can take parameters as input and return one value as their output.

Listing 4.12: A VDM class with two operations.

```
class Sensor
operations
public getValue : () ==> int
getValue() == return 1;

public setId : int ==> ()
setId(id) == skip;
end Sensor
```

Figure 4.13: Sensor with two operations. The isQuery property is hidden.

Listing 4.12 shows a VDM class Sensor with two operations getValue with a return parameter int and setId with a parameter id of the type int. The operations are mapped to a UML class shown in figure 4.13 the property isQuery is not visible which is the property differencing a operation from a function.
Transformation Rule 17
A VDM operation and function are mapped to the UML meta-class Operation where the property isQuery determine whether the Operation represents a VDM function or operation:

- **true** for a function.
- **false** for a operation.

The return type of a function and operation is mapped collectively as the property type and the multiplicity\(^3\) of the Operation meta-class. The parameters of the operation or function is mapped to the UML meta-class Parameter represented as the property ownedParameters of the Operation meta-class. The name and type of a VDM parameter are mapped to the property name, type and the multiplicity\(^3\) of the Parameter meta-class.

\(^3\)The multiplicity consists of the properties isOrdered, isUnique, lower and upper.
Chapter 5

Static Model Specification

This chapter describes how the transformation rules from chapter 4 are turned into a VDM model. The chapter begins with an overview of the structure of the static model transformation. The Abstract Syntax Tree (AST) of UML and related tools are presented along with a description of how the specification enables a transformation from a VDM model to a UML model and vice versa. This is followed by a description of how to merge an abstract VDM and UML model without information loss.

5.1 Abstract Syntax Tree

An AST is an abstract structure which contains only core information, i.e. the VDM AST do not contain keywords or tokens used in the concrete syntax. The reason is that the concept of an instance variable is defined as a node in the tree, hence the member declaration group instance variables do not provide extra information.

The UML AST is inspired from the OML AST (the AST for VDM by Overture), and as such it consist of VDML-SL type definitions for each UML meta-class. Both the OML and UML AST can be populated with data originating from source files containing concrete information. For example, the OML AST can be populated using the Oml Parser, as shown in Figure 5.4, which parses VDM class files and returns the OML AST with a specification of the total model. The UML AST can be populated in an similar way by parsing a UML Model file (XMI).

5.1.1 AST definition to VDM class structure (ASTGen)

Using the types of an AST in a VDM model requires information about the structure of the AST in order to convert collections of type definitions into classes. The classes can subsequently be instantiated and populated with concrete information. The Overture tool ASTGen enables the transformation of a collection of type definitions into a class
structure. It also generates a visitor pattern and creates Java interfaces that correspond to the abstract VDM classes used in the visitor pattern. The implementation of the visitor pattern makes it possible to add functionality to visit the AST by adding a virtual function to the classes in the AST. The tree can then be visited without changing the AST, just by the use of a reference to the AST [VisitorPattern]. The Java interfaces provides the ability to implement the instantiated AST directly into Java code. The tool works by taking a type specification as input in addition to information like a prefix and a output location also are provided.

![Figure 5.1: ASTGen with UML.ast as input.](image)

ASTGen takes an AST file as input containing the AST for a language. For each type specified in the AST, e.g. B from listing 5.1, ASTGen produces the following:

**Two VDM classes:** One abstract class and one implementation of the abstract class per type definition: IUmlA, IUmlB, IUmlC with get and set operations for instance variables, UmlA, IUmlB and UmlC which inherits the corresponding IUML class and holds instance variables.

**One Java interface:** Corresponding to the abstract class mentioned above.

**A visitor:** An implementation of the Visitor pattern for visiting the each kind of node in the AST.

Listing 5.1 shows the AST type definitions. This definition defines A, B and C where B contains a name. All definitions are listed as VDM-SL type definitions. In the top of the file additional information for ASTGen such as prefixing and output location is located in addition to a package name and root node. Inheritance is generated from the union operator |. In this case A = B | C will be equivalent to B and C inheriting A.
5.1. ABSTRACT SYNTAX TREE

Listing 5.1: AST definition used as input for ASTGen.

In Listing 5.2 the VDM interface class prefixed with IUML is shown which is similar to the Java interface also generated.

```vdm
class IUmlB
  is subclass of IUmlA

  operations
  public getName: () ==> seq of char
  getName() == is subclass responsibility;

end IUmlB
```

Listing 5.2: The VDM TemplateParameter interface class derived from the type defined in listing 5.1.

```vdm
class UmlB is subclass of IUmlB

  operations
  public identity: () ==> seq of char
  identity () == return "B";

  public accept: IUmlVisitor ==> ()
  accept (pVisitor) == pVisitor.visitB(self);

  ... instance variables
  private ivName : seq of char := []

  operations
  public getName: () ==> seq of char
  getName() == return ivName;

  public setName: seq of char ==> ()
  setName(parg) == ivName := parg;

end UmlB
```

Listing 5.3: The implementation of B defined in listing 5.1.
In Listing 5.3 the implementation class of the $B$ type definition from listing 5.1 is shown. This class can be used in a specification. Additional to the interface and implementation classes a visitor is generated at the VDM level.

ASTGen is limited by the Java language since it has to create Java interfaces for each type definition. As a result of the close bounding to Java ASTGen is limited to single inheritance. It is possible to specify multiple inheritance as shown in 5.4. Figure 5.2 shows how the types definitions from Listing 5.4 results in multiple inheritance which ASTGen cannot resolve, since it violates the Java language specification.

```
A = B | C;
D = B | E;
B ::
    name : seq of char;
C ::;
E ::
```

Listing 5.4: AST definition with multiple definition. Not supported by ASTGen.

![Diagram showing multiple inheritance](image)

Figure 5.2: Shows multiple inheritance. $B$ inherits $D$ and $A$.

### 5.1.2 OML AST

The AST for VDM is already made and is available as a part of the Overture project [Overture07]. The abstract syntax is named Overture Modeling Language (OML) and it is specified as type definitions in VDM-SL, as shown in Listing 5.5. The entire AST can be found in Appendix G. The type definitions in the OML AST can be converted to VDM++ classes, VDM++ interface classes and Java interfaces using ASTGen as explained in section 5.1.1 above. The resulting VDM++ classes can be used when modeling in VDM++. The Specifications is the root node and it contains a sequence of classes which is specified as the next node. The node Class consists of new nodes specified further down in the OML AST.
5.1. ABSTRACT SYNTAX TREE

Specifications ::
    class_list : seq of Class;

Class ::
    identifier : Identifier
    generic_types: seq of Type
    inheritance_clause : [InheritanceClause]
    class_body : seq of DefinitionBlock
    system_spec : bool;

InheritanceClause ::
    identifier_list : seq of Identifier;

DefinitionBlock =
    TypeDefinitions |
    ValueDefinitions |
    FunctionDefinitions |
    OperationDefinitions |
    InstanceVariableDefinitions |

Listing 5.5: Abstract syntax of VDM. Overture Modeling Language (OML).

5.1.3 UML AST

The UML AST is inspired by the OML AST and constructed from the UML specifications [UMLInfrastructure2.1.2, UMLSuperstructure2.1.2], hence it is a flat tree structure with a root node, comparable to the Specifications node of the OML AST from Listing 5.5. The UML AST is modeled as VDM-SL type definitions to enable automatic model-generation using the tool ASTGen as explained in section 5.1.1. The type definitions are consistent with definitions from the UML Superstructure Specification (USS) [UMLSuperstructure2.1.2] and UML Infrastructure Specification [UMLInfrastructure2.1.2] (UIS), which makes intensive use of multiple inheritance that is not supported by ASTGen as explained in section 5.1.1. To change the structure into single inheritance some of the abstract meta classes e.g. Classifier of the USS and UIS has been replaced by a specific class representing the classifier (Classifier is an abstract base class of Class).

Figure 5.3 shows a condensed class diagram from the UIS. The excerpt show the connections between the UML meta-classes which is used in the UML AST e.g. Class, Operation, Property etc. as explained on page 150.

The UML AST is constructed from the meta-classes needed in a class diagram. This means that nodes like: classes, operations, properties, associations, constraints etc. are considered as nodes in the UML AST. They all have a corresponding meta-class in the USS and UIS. In Listing 5.6 an example is given of the top level of the AST. The root
node named `Model` is specified and it consists of a name and a set of definitions which is represented by model elements such as `Class`, `Association`, etc.

\[
\text{Model} ::=
\begin{align*}
\text{name} & : \text{String} \\
\text{definitions} & : \text{set of ModelElement};
\end{align*}
\]

\[
\text{ModelElement} = \text{Class} \mid \text{Association} \mid \text{Constraint} \mid \text{Collaboration};
\]

Listing 5.6: UML toplevel AST

The complete UML AST is specified in appendix E along with a description that states where each node in the UML AST complies with the USS and UIS.

### 5.2 Transformation Specification Overview

From a user's point-of-view, the model transformation between VDM and UML occurs on the concrete syntaxes. A common approach when working with constructs in a computer-based language, is to use an AST which defines the abstract syntax (i.e.
without tokens etc.). ASTs are often used when writing compilers, because it provides an easy way of accessing different constructs in a language [ASTFromWikipedia]. In the case of UML, an AST reduced the amount of information because the concrete syntax is presented in XML, which produces a lot of overhead. The UML AST defined as part of this thesis work can be found in Appendix E. The OML AST is located in Appendix G. The transformation rules specifying how to apply a transformation between the two ASTs are defined in chapter 4.

5.3 The Transformation Process

This section presents the transformation process that can be applied to a VDM and UML model. Figure 5.4 gives an overview of the steps involved when applying a transformation. The figure is supplemented with a walk-through for each transformation direction: (1) Transforming VDM to UML and (2) Transforming UML to VDM.

![Diagram of transformation process](image)

**Figure 5.4: Overview of components involved in the transformation process.**

In Figure 5.4 The solid lines show how a transformation from VDM to UML is carried out. The dotted line shows how an UML model (stored in an XMI file) is transformed into VDM class files. The rectangular shapes indicate processing and the round shapes indicates a processing result. The grey rectangular shapes indicate components written in Java. All other rectangular shapes indicate components written in VDM.

The transformation from VDM to UML consists of the following steps. Each step is supplemented with a note describing the status immediately after the step:

**Step 1 (Oml Parser):** The VDM classes are parsed using the OML Parser to a VDM abstract syntax, i.e. the parser populates the OML AST.
CHAPTER 5. STATIC MODEL SPECIFICATION

- **Status**: An OML AST existed before this thesis work started. It is comparable to the UML AST, thus making it possible to compare or transform between the two.

**Step 2 (Vdm2Uml)**: The OML AST is transformed into an UML abstract syntax using the `Vdm2Uml` specification (which is consistent with the transformation rules from chapter 4).

- **Status**: The UML abstract syntax is ready to be transformed to a XML structure.

**Step 3 (Uml2Xmi)**: The UML abstract syntax is transformed into a XML structure using the `Uml2Xmi` specification\(^1\).

- **Status**: The XML data required to produce valid XMI is completed.

**Step 4 (IO)**: The XML data is processed and output is a XML document formatted to comply with the rules for valid XMI structure.

- **Status**: The XML document which can be imported to a UML 2 and XMI 2.1 compliant tool.

The transformation of an UML model to VDM class files is shown in Figure 5.4 as dotted arrows. The following steps explains what happens.

**Step 1 (Xmi Parser)**: The XMI file representing the UML model is transformed into a corresponding XML document (`Xml Model`).

- **Status**: The XML data representing the UML model is ready to be transformed to UML abstract syntax.

**Step 2 (Xml2Uml)**: The XML data is transformed into UML abstract syntax, i.e. the UML AST is populated.

- **Status**: A UML abstract syntax exist, which is comparable to the VDM abstract syntax.

**Step 3 (Uml2Vdm)**: The UML abstract syntax is transformed into a VDM abstract syntax using `Uml2Vdm`\(^2\).

- **Status**: The VDM abstract syntax is ready to be visited by Oml2VppVisitor.

**Step 4 (Oml2VppVisitor)**: Every construct of the VDM abstract syntax is visited by Oml2VppVisitor, which outputs a VDM class file equivalent to the UML model\(^3\).

- **Status**: A VDM class file equivalent to the UML model has been produced.

\(^1\)If modeling tools require the XML file to be tool specific this component needs to change and if one wants to be able to produce different XMI for different UML tools it would be done here.

\(^2\)Only basic constructs are transformed.

\(^3\)Only the VDM constructs supported by the transformation in this thesis work.
5.4 Transforming VDM to UML

The transformation of an abstract VDM model into a corresponding abstract UML model is modeled in the class `Vdm2Uml` and based on the transformation rules from chapter 4. `Vdm2Uml` is called with an `OmlSpecification` which is the root in the AST for VDM. It then walks down the tree and gathers information used to create a `UmlModel` which is the root of the UML AST.

The interesting part in the transformation of a OML class is the `class.body` which can consist of multiple definition blocks, e.g. `types`, `values`, `instance variables`, `functions`, `operations` and `thread` as shown on page 71. All these definition blocks have to be treated individually since they require a specific transformation, based on their type. The `ValueDefinitions` and `InstanceVariableDefinitions` have the most significant impact on the visual part of the static structure, since they present properties and associations that is directly visible in a class diagram. Both blocks are transformed into UML properties of the corresponding classes or associations according to rule 6 and rule 5.

The main path through the `OmlSpecification` is first to find all classes that can be mapped to UML Classes. When a class is found, it is transformed into a UML class using the transformation rules from chapter 4. The values and instance variables of the class are transformed into a corresponding structures in UML. When an `InstanceVariableDefinitions` of a data type is found in the body of an OML class, the definition is transformed into a UML property. The created property will be associated with the owning class as an attribute according to rule 6 on page 56. When an instance variable is a class reference type it is transformed into an association, according to rule 5 on page 55. See Figure 5.5 for an overview of the operations involved in the transformation of a class.

If the instance variable or value represents a product type, union type or map type, the created property needs additional transformation according to rule 10 on page 58, rule 9 on page 57 and rule 12 on page 62. Instead of being transformed into a property of the class, they are transformed into associations. An example is the union type explained in rule 9 on page 57. The union type is a composite type, which can be one of two types specified in the union, which means that apart from creating the association additional information needs to be added to the UML model to ensure that only one of the specified parts in a union type can exist at the time.
CHAPTER 5. STATIC MODEL SPECIFICATION

```java
public init : IOMlSpecifications --> IUmlModel
init(specs) ==
{ let model = build_uml(specs)
  in
  ( model.setDefinitions(model.getDefinitions()
    union associations
    union constraints);
    return model;
  );
};
```

Listing 5.7: Vdm2Uml init operation.

Figure 5.5: Overview operations/ functions involved the transformation.

The top level of the transformation specified in the Vdm2Uml class is shown in Listing 5.7 and its hierarchy position is shown in Figure 5.5. The steps involved in the transformation are described below:

- First, the `build_uml` operation jumps one level down in the tree to construct the classes.

- Each class is constructed by `buildClass`, which again jumps one level down to construct the body of each class. When constructing the body of the classes all associations are deduced from instance variables and values definitions.

- The instance variables, association and constraints shown in line 6 and 7, are populated from `buildVariable`(`instance variables`) and `buildValue`(`values`) seen in Figure 5.5. The caller of this operation is the `buildClass` operation.
After the creation of all classes from the OML specification, a UML model is created from the UML classes and the associations created from values and instance variables from the OML specification along with the constraints found from union types.

### Listing 5.8: Abstract syntax of an OML class.

```
Class ::
    identifier : Identifier
    generic_types : seq of Type
    inheritance_clause : [InheritanceClause]
    class_body : seq of DefinitionBlock
    system_spec : bool;
```

### Listing 5.9: Abstract syntax of an UML class.

```
Class ::
    name : String
    classBody : set of DefinitionBlock
    isAbstract : bool
    superClass : seq of ClassNameType
    visibility : VisibilityKind
    isStatic : bool
    isActive : bool
    templatesignature : [TemplateSignature];
```

In Listing 5.8 and 5.9, the OML and UML representation of a class is shown. This is the first node that needs to be transformed. By the use of rule 1 on page 53 the class can be transformed into an UML class. Since all classes in VDM exists as public classes the corresponding UML class will have its visibility set to public.

- **Class name**: The name of the UML class can be extracted from the OML class identifier and the super classes for a UML class can be deduced from the inheritance clause.

- **Abstract or Active**: To deduce if a class should be mapped as abstract or active it is necessary to look inside the body definition of the OML class, if an operation is delegated to a subclass, it is an abstract class or if a thread definition exists the class is mapped as an active class.
**public build Class : IOmlClass --> IUmlClass**

**build Class(c) ==**

\[
\text{let name} = c.getIdentifier(), \\
\text{inh} : [IOmlInheritanceClause] = \text{if c.hasInheritanceClause() then c.getInheritanceClause() else nil,} \\
\text{body} = c.getClassBody(), \\
isStatic = false, \\
isActive = card \{ \text{body(i)} | \text{i in set inds body} \} > 0, \\
dBlock = [\text{let dbs : IOmlDefinitionBlock = body(i) in build_def_b(dbs,name)} | \text{i in set inds body}], \\
dBlockSet = \{ d | d \text{ in set elems dBlock & d <> nil}, \\
isAbstract = \text{hasSubclassResponsibilityDefinition(body)}, \\
supers = \text{getSuperClasses(inh)}, \\
visibility = \text{new UmlVisibilityKind(UmlVisibilityKindQuotes'IQPUBLIC)}, \\
templateParameters = \text{getGenericTypes(c.getGenericTypes())} \}
\]

\[
\text{in} \\
\text{return new UmlClass(name,} \\
dBlockSet, \\
isAbstract, \\
supers, \\
visibility, \\
isStatic, \\
isActive, \\
templateParameters); \\
\]

Listing 5.10: Vdm2Uml operation for constructing an UML class.

The `build Class` operation shown in listing 5.10 is called from the `init` operation with all the classes found in the OML specification. The `init` operation is the entry point at the transformation. The operation builds a `UmlClass` from a `OmlClass`. The interesting part is how the body of the OML class contributes to the UML class. The UML class is declared `active` if a thread definition is found in the body of the OML class at line 9, and `abstract` if an operation is found that is subclass responsibility at line 15. To construct the attributes of the class the values and instance variables definitions are handled in line 11. The `buildVariable` operation is shown in Listing 5.11. This operation is responsible for the transformation of instance variables and is called indirectly\(^4\) from `build Class` shown in figure 5.5.

---

\(^4\) The creation of instance variables are indirectly called through `build_def_b` which is a utility function required by the code generator. The `build_def_b` redirects the call to `buildVariable`. 

public buildVariable : IOmlInstanceVariable * String ==> [IUmlProperty]
buildVariable(var,owner) ==
  let
      access  = var.getAccess(),
      scope  = access.getScope(),
      assign = var.getAssignmentDefinition(),
      isStatic = access.getStaticAccess(),
      name = assign.getIdentifier(),
      visibility = convertScopeToVisibility(scope),
      omlType = assign.getType(),
      multiplicity = Vdm2UmlType’extractMultiplicity(omlType),
      type = Vdm2UmlType’convertPropertyType(omlType,owner),
      isReadOnly = false,
      default : [String]= if assign.hasExpression() then
        getDefaultValue(assign.getExpression())
        else nil,
      isComposite = false,
      isDerived = false,
      qualifier : [IUmlType] = Vdm2UmlType’getQualifier(omlType)
  in
    ( ...
    if not isSimpleType(omlType)
    then
      ( CreateAssociationFromProperty(property,omlType);
        return nil
      )
    else return property; );

Listing 5.11: Creates a property from an instance variable or an association depending on the type of the instance variable.

When a property is constructed, the OML type of the property decides whether it should be inserted into the UML model as an attribute of the owning class, or as an association-end of an association. The decision is based on rule 5 on page 55. This means that types like class, maps, union types and product types are all mapped as associations. If a property is of one of the abovementioned types, it must map as an association. Depending on the type of the property, the operation CreateAssociationFromProperty is called with the property and the original OML type:

- **Product Type**: CreateAssociationFromPropertyProductType
- **Union Type**: CreateAssociationFromPropertyUnionType
- **All other types**: CreateAssociationFromPropertyGeneral

When an instance variable of a product type is discovered it must be converted into an association. This causes the CreateAssociationFromProperty to delegate the
construction to CreateAssociationFromPropertyProductType (see listing 5.12). Here, an association is created and stored in the Vdm2Uml class.

```
public CreateAssociationFromPropertyProductType: 
  IUmlProperty * IOmlType --> ()
CreateAssociationFromPropertyProductType(property,omlType) ==
let name : String = property.getName() ,
  prop : UmlProperty = property
props : set of IUmlProperty =
  dunion |CreateEndProperty(p,name)
  | p in set {omlType}
  & isoofclass(IOmlProductType,p})
in
  { prop.setName(""); 
if card props > 1 then
  associations := associations union
    {new UmlAssociation(props, {prop},nil,GetNextId())}; }
```

Listing 5.12: Create an association from a Product type.

The association is constructed according to rule 10 on page 58 where each part of the product type is represented as an end in the association. To achieve this, the product type and the property (instance variable) name are passed to CreateEndProperty (see listing 5.13).
TRANSFORMING VDM TO UML

<table>
<thead>
<tr>
<th>Left type</th>
<th>Right type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductType1</td>
<td>A * B * C</td>
</tr>
</tbody>
</table>

Table 5.1: Product type constructed of three types.

```java
public CreateEndProperty : IOmlType * String --> set of IUmlProperty
CreateEndProperty(t,name) ==
{
  if (isofclass(IOmlProductType,t)) then
    let typedType : IOmlProductType = t
    in
    return CreateEndProperty(typedType.getLhsType(),name)
    union CreateEndProperty(typedType.getRhsType(),name);
  else if (isofclass(IOmlUnionType,t)) then
    let typedType : IOmlUnionType = t
    in
    return CreateEndProperty(typedType.getLhsType(),name)
    union CreateEndProperty(typedType.getRhsType(),name);
  else
    return {new UmlProperty(name,
                            new UmlVisibilityKind(
                                UmlVisibilityKindQuotes'IQPRIVATE),
                                Vdm2UmlType'extractMultiplicity(t),
                                Vdm2UmlType'convertType(t),
                                ...
                                Vdm2UmlType'getQualifier(t))};
}
```

Listing 5.13: Create association ends property from a OML type.

The construction of association-end properties for both product and union types are shown in Listing 5.13. The operation CreateEndProperty is recursive in the sense that it keeps calling itself until it reaches the base case, which is defined such that the `t` at line 6, product type passed to the function is broken down in its left and right side and when it is no longer a product type the operation terminates and returns a new property for each recursion.

An example of such a product type can be seen in Table 5.1. The product type in Table 5.1 is constructed as two product types where the last `ProductType2 = (B * C)` is contained in the first `ProductType1 as (A * ProductType2). When this ProductType1 = A * B * C is put into CreateEndProperty (see listing 5.13) it will recognize the ProductType1 in the first recursion. CreateEndProperty will then first resolve the right type, which again is a product type leading to the first recur-
sion. After this type is resolved to two types \((B \times C)\) it will create properties from the \(B\) and \(C\) type. When returned the properties will be combined with the property created from \(\text{ProductType1}\) left side of the \(A\) type. The properties are then tied together into an association as non-navigable ends.

The transformation of collections are done by manipulating the multiplicity of the constructed properties. Listing 5.14 shows a snippet of the model setting the multiplicity for \(\text{set}, \text{seq}, \text{seq1}, \text{map}\) and \(\text{imap}\). The multiplicity element is set according to rule 11 on page 60.

```plaintext
cases true:
(\text{isofclass}(\text{IOmlSetType},t))\rightarrow
(lower := 0 ; upper := \text{nil}; \text{isOrdered} := \text{false}),
(\text{isofclass}(\text{IOmlSeq0Type},t))\rightarrow
(lower := 0 ; upper := \text{nil}; \text{isOrdered} := \text{true}; \text{isUnique} := \text{false}),
(\text{isofclass}(\text{IOmlSeq1Type},t))\rightarrow
(lower := 1 ; upper := \text{nil}; \text{isOrdered} := \text{true}; \text{isUnique} := \text{false}),
(\text{isofclass}(\text{IOmlGeneralMapType},t)),
(\text{isofclass}(\text{IOmlInjectiveMapType},t))\rightarrow
(lower := 0 ; upper := \text{nil}; \text{isOrdered} := \text{true}; \text{isUnique} := \text{false}),
(\text{isofclass}(\text{IOmlOptionalType},t))\rightarrow
(upper := 1 ; lower := 0)
end;
```

Listing 5.14: Setting multiplicity of properties.

Listing 5.14 shows an example of extracting the multiplicity. A \(\text{set}\) results in a multiplicity with no upper limit and a lower limit of zero. There are no ordering and all elements are unique, which corresponds to \(\text{isUnique}=\text{false}\).

### 5.5 UML Model to XMI

To transform the UML AST into the XML Metadata Interchange (XMI)\(^3\) an abstract model of a XML file has been made named XML API. This is a simple model structure that represents an XML document, where elements can have other elements, attributes and data. This abstract model is then implemented with a visitor pattern that enables easy implementation of a visitor to print the model to a file stream. In addition to this, a parser has been modeled to ease the process of parsing an existing file into this abstract model. The structure of the XML API can be seen in figure 5.6.

\(^3\)OMG standard for exchanging metadata information via Extensible Markup Language (XML) used to serialize UML models to different UML modeling tools.
5.5. UML MODEL TO XMI

Transforming the UML AST into an XML abstract model is trivial in principle. The only added information is an identifier for each element and the type of the element. The identifiers are used to relate elements such as the type of a property, qualifiers of associations and constraints of an association.

Listing 5.15: XML Schema of Class representation in a XMI document specified in MOF.

In Listing 5.15 an XML Schema is shown specifying the rules for inserting a class into a XMI document where `name="c"` is the name of the class and `type="c"` corresponds to the complex type defined at line 2. At line 3-5 the optional `xmi:Extension` is listed. The attribute reference at line 7 indicated that the attributes from `xmi:ObjectAttribs` should be included. The description originates from the MOF XMI specification [MofXmi, p17]. The schema has included both a name space for UML elements and a name space for XMI elements:

- **uml**: http://schema.omg.org/spec/UML/2.0
- **xmi**: http://schema.omg.org/spec/XMI/2.1

Figure 5.6: Overview of the XML API class structure.
In Listing 5.16 a single class is inserted into an XMI document by the aid of the schema specified in listing 5.15. The class is named C1 and has an attribute named a1 with a private visibility. The same schema for both classes, properties, associations etc. can be applied the nodes specified in the UML AST. Listing 5.17 shows a snippet of the process of converting a UML AST Class into a XMI document. The process of converting the nodes from the UML AST into a XMI document is straightforward.

As seen in Listing 5.17, the different information from the UML AST class is mapped into specific named attributes, e.g. the name of the class in line 7.

### 5.5.1 XML parser / deparser

To enable a round-trip between a the UML AST and the XMI document a parser need to be introduced along with a deparser to transform an XML document into the UML AST.

- XML parser to populate the abstract model of the XML document.
- XML deparser (Xml2Uml) enabling the transformation from the abstract XML document into the UML AST. This is the reverse process of transforming the UML AST to XMI done by Uml2Xmi.

### 5.6 Transforming UML to VDM

Transforming a UML document into a VDM model is the reverse process of VDM to UML. If a transformation from UML to VDM is desired, then all the associations and
their constraints have to be converted into values and instance variables so they can be attached the owning class when it is transformed back from a UML class. In Listing 5.18 the initial operation for the UML to VDM transformation process is shown.

```plaintext
public init : IUmlModel ==> IOmlDocument
init(model) ==
let
associations = { a | a in set model.getDefinitions()
  & isofclass(IUmlAssociation,a)},
constraints = { a | a in set model.getDefinitions()
  & isofclass(IUmlConstraint,a)}
in
{ extractInstanceVarsFromAssociations(associations,constraints);
  return new OmlDocument(model.getName(),
    new OmlSpecifications(build_classes(model)),[]); }
```

Listing 5.18: Setting multiplicity of properties.

Listing 5.18 shows how the associations and constraints are first extracted from the model definitions. Then all instance variables and values represented as associations are extracted (see line 10) and saved in a local map linking a class name to the instance and value definitions found in the associations. In Figure 5.7 an overview tree is shown of the Uml2Vdm class. It lists the placement of the operations and functions described in this section.

Figure 5.7: Overview of the operations and functions described in this section.

```plaintext
public extractInstanceVarsFromAssociations : 
  set of IUmlAssociation *
  set of IUmlConstraint ==> ()
extractInstanceVarsFromAssociations(associations,constraints) ==
let product = -- N-ary association to single product type
```

6The local map is not described in this chapter. It is named classInstanceVars and further details can be found in Appendix F.
CHAPTER 5. STATIC MODEL SPECIFICATION

Listing 5.19: Extract all associations that represents a product type.

Listing 5.19 shows how an association representing a product type is distinguished from other associations by the fact that it does not have an xor constraint and more than two ends. Such associations must have at least three ends: One for the class owning the product type and at least two others to construct the product type. When a N-ary association is found, the product type is extracted (line 14-16).

Listing 5.20: Extract the association end that owns the product type, convert the other ends into a product type and store it in a map linked to the owning class.

In listing 5.20 two important things occur: (1) the owner of the product type, the class where the product type should be placed are found and (2) the product type is created from the remaining ends of the association. Only the types of these ends are needed line 11. The product type is created at line 13 and then added to the owning class through a class name to definition map at line 15 through the CreateProductType operation. This makes the operation create the final class definition to include the definitions extracted as associations from e.g. product types.
5.7 Merging Changes in VDM and UML Models

Merging a VDM and UML model which both differ, leads to problems such as determining which is the one that overrules the other. In this section the problem of merging will be discussed and a solution will be proposed based on the transformation described earlier in this chapter.

Before a merging process can be completed it is required that the structures that should be merged are comparable. In this case where an abstract model of VDM and UML should be compared, it is required first to bring them on the same form. This leads to the question if the OML AST or UML AST should be used as common base. The UML side would be preferred since the transformation VDM to UML contains most features at the time of writing.

In Figure 5.8 a merging process is shown. The VDM model is transformed into an abstract UML model and then compared with the existing UML model. The result of this comparison is a change log. This change log is a description of what has changed and the last modified date-time. The information leads to an estimate of which direction the classes should be mapped.

A problem occurs, if an operation depends on an instance variable and both change, but in different models, i.e. the instance variable changes in the UML model and the operation changes in the VDM model. This leads to multiple merging problems because the operation is depended on the instance variable. There are two solutions to this issue (1) Search the OML AST and find the operation, decide whether the instance variable is used: if not then merge, if it is possible without any problems. (2) A single class is

```plaintext
private CreateProductType : seq of IUmlType --> IOmlType
CreateProductType(tps) ==
  let first = hd tps,
      rest = tl tps,
      front = ConvertType(first)
  in
    if len tps = 1 then
      return front
    else
      return new OmlProductType(front, CreateProductType(rest))
pre len tps > 0;
```

Listing 5.21: Create one product type from the types associated with the association ends.

The operation in Listing 5.21 is a recursive function that makes a recursion for each type in `tps`. Each time the head of the list is chopped off (line 3) and the current type is converted to the correct VDM type. If a rest exists (rest line 4), it is parsed to the function itself creating a product type of the rest (line 10). When done one product type has been created from all the types passed to the operation.

5.7 Merging Changes in VDM and UML Models
declared changed if any part of it is changed and only complete classes can be merged from one model to another. When the change log is created and the user has decided which classes should be merged in a certain direction, a merge can take place. However, it is important to use the original OML AST when updating the VDM source, because this thesis work only has incorporated a subset of the available VDM constructs. Using the original OML AST avoids information loss. If only the OML AST is updated or parts removed, no information will be lost and the changes from the UML model will be applied to the OML AST. The final part would then be to backup the original files and write the new merged files.
Chapter 6

Interaction Model Transformation

In this section, VDM traces are related to UML 2 Sequence Diagrams (SDs) through model transformation. A significant extension to UML 2 Sequence Diagrams (SD) is the CombinedFragment which permit the expression of procedural logic in Sequence Diagrams. It is also possible to nest fragments to an unlimited degree. The expression of procedural logic in UML 1 SDs was possible using labels on messages, stating a condition and/or iteration expression. However, An SD could quickly lose its clarity with too many messages labeled with various conditions and iterations. The Combined-Fragments of UML 2 mitigates the problem by surrounding entire sets of messages and supplying a common continuation predicate. In this chapter a description of how UML Sequence Diagrams (SD) and VDM traces can be combined used in such a way that they presents the same content is given in section 6.1 which illustrates the possible connection. Then rules are formed in chapter 6.2 clarifying how specific constructs can be transformed.

The following uses of SDs in conjunction with VDM traces may be utilized: The Unified Modeling Language 2 (UML) contains a dynamic view SD for presenting a dynamic interaction between objects of a system. A SD consists of Lifelines which presents an instance of a class in the system, Messages which presents the interaction between objects, CombinedFragment together with InteractionOperand and its associated Constraint presents a constraint execution of messages. A Combined-Fragment is in itself a container as the SD which means that nesting are possible it self two kinds of CombinedFragments are used alt and loop see section 2.6.2 for more information. The guard of an InteractionOperand the Constraint constrains the execution of messages associated with it. Practically, to avoid bloating nesting of SD should be limited to one level only. There by denying nesting of SD inside a Combined-Fragment.
The following uses of Sequence Diagrams in conjunction with VDM may be utilized:

**Traces generation:** VDM test cases may be generated automatically from traces, i.e. compressed representations of test cases. Sequence Diagrams may be utilized to visualize the resulting test cases from a trace.

**Model generation:** A specific interaction among objects may be specified by a Sequence Diagram and generated the traces part of a VDM model. The interaction among objects may represent a fragment of a model or test cases.

**Diagram generation:** The interaction of objects of the traces part of a VDM model may be visualized as a Sequence Diagram.

To meet the new traces feature of VDM introduced to enable easy regression testing SDs is regarded as having the greatest value for VDM modelers. By the use of SD both VDM specialists and ordinary software developers with UML experience will be able to specify test cases of a model. By transforming SD into a trace statement a trace can be seen both as a VDM statement or as a visual UML diagram. In this chapter the focus will be to construct rules to enable a transformation between both of them. From the rules round trip would be possible. Thus, this thesis pursues the application of Sequence Diagrams as means for an overview of test cases and the creation of VDM traces from the visual diagram.

### 6.1 VDM traces and UML sequence diagrams

VDM traces is an advanced way of specifying a sequence of execution (see section 3.6) and from this point of view a UML sequence diagram must be able to show the same execution. In figure 6.1 a SD is shown where the corresponding traces statement is specified in listing 6.1. It can be seen that `Messages` relates to the `MethodApply` expression in the traces statement and the `Lifelines` refer to the objects involved in the execution.
6.2 Transformation Rules

The transformation rules for an interaction transformation describes, how instances of certain UML meta-classes are related to constructs of the VDM concrete syntax regarding Trace Definitions [LangManPPTraces]. The syntax definition can be found in section 3.6 on page 49.

6.2.1 Trace placement

The placement of a trace statement is derived from LifeLines of a SD. To deduce which LifeLine should be the owner of a trace statement a closer look at the Messages of the SD is needed. Messages consists of a sendEvent and sendReceive representing a MessageEnd which is of type MessageOccurenceSpecification Mos for short. Each Mos covers one LifeLine. The owner of the trace statement is the LifeLine where Messages only origin from.

Figure 6.2 shows the meta-classes involved when deducing the connection between Message sendEvent and sendReceive by a Lifeline.
CHAPTER 6. INTERACTION MODEL TRANSFORMATION

Figure 6.2: VDM Meta-classes showing the link between Message and LifeLine.

Figure 6.3: Sequence diagram.

Listing 6.2: Operation to get owner of a trace from the messages involved.
6.2. TRANSFORMATION RULES

**Transformation Rule 18**
The class where the trace is placed is the one from which all Messages in a SD originates.
An interaction transformation is only possible if all Messages origin from a single Lifeline.

6.2.2 Trace name

A named trace definition gets it name from its counterpart in UML Interaction which is the Meta-class holding all Meta-classes conforming a sequence diagram. It is the property Name of the Interaction Meta-class that holds the name that can be transformed to the name of a named trace.

![Sequence diagram with a name](image)

**Listing 6.3:** Class with a named trace acceding to Figure 6.2.2.

Transformation Rule 19
The attribute Name of the UML meta-class Interaction is mapped as the name of the trace.

6.2.3 Trace Apply Expression

The trace apply expression contains information about which object a certain method should be executed on and value for the parameters of the method. This information is transformed from the SD mainly by the Meta-class Message by the aide of Mos, Lifeline and CallEvent. Where the Lifeline represents the objects that the method should be executed upon and the CallEvent references the method that should be executed through the Mos to the Message.
CHAPTER 6. INTERACTION MODEL TRANSFORMATION

Transformation Rule 20
The method name in a trace apply expression is transformed from the Operation property of the Meta-class CallEvent and the variable on which the method should be executed is transformed from the LifeLine at the receive end of the message where a Mos is linking it to a LifeLine representing the object. The arguments are directly transformed from the Message Meta-class.

6.2.4 Sequencing of trace apply expressions
The apply expressions in a trace is transformed from the SD in the order they are specified in the SD. This is a one to one transformation between the ordering of Messages in a SD and the order of method apply expressions in a trace.

Transformation Rule 21
Method apply expressions in a trace are sequenced in the same order as messages in a SD.
6.2.5 Trace choice operator

Messages from a SD is transformed into method apply expressions separated by the choice operator if they are contained in the same CombinedFragment and placed in an operand each where the CombinedFragment InteractionOperator equals alt.

![Sequence diagram with loop fragments](image)

Figure 6.7: Sequence diagram with loop fragments.

```
class TraceChoiceEx
  traces
    TraceChoiceOperator :
      b.Message1() | b.Message2()
  end TraceChoiceEx
```

Listing 6.6: Class with a named trace showing the choice between two apply expressions.

---

Transformation Rule 22

Messages are transformed into method apply expressions separated be the choice if they are contained in the same CombinedFragment in each their operand where the CombinedFragment InteractionOperator equals alt.

---

6.2.6 Repeat Pattern for apply expressions

The repeat pattern is transformed from the UML Meta-class Operand holding the message, where the property InteractionConstraint decides the repeat pattern according to table 23 and the CombinedFragment where the Operand is contained in having its InteractionOperator set to loop.
Figure 6.8: Sequence diagram with \texttt{alt} fragment.

Transformation Rule 23
The repeat pattern of a apply expression is transformed from the InteractionConstraint of an Operand contained in a CombinedFragment where the InteractionOperator equals \texttt{loop}. The constraint of the Operand holding the message specifies how the repeat pattern should be set:

<table>
<thead>
<tr>
<th>Constraint (Guard)</th>
<th>RepeatPattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a*}</td>
<td>\texttt{a+}</td>
</tr>
<tr>
<td>minint</td>
<td>0</td>
</tr>
<tr>
<td>maxint</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 6.1: Transformation rules for VDM constructs modeling collections

6.2.7 Nested sequencing messages
Messages are transformed into method apply expression according to their order and they are grouped together in brackets with messages that exists in the same CombinedFragment where the InteractionOperator equals \texttt{loop}. It is allowed to have CombinedFragment’s inside other CombinedFragment’s which enables a Message to be both optional and have a repeat pattern. The Mos associated with a message must only be associated with one operand in one CombinedFragment.
Figure 6.9: Class with nested messages in Combined fragments.

Listing 6.8: Class with nested method apply expressions.

Transformation Rule 24

Messages nested in a fragment of an InteractionOperand is added the same properties as Messages in the parent(s) InteractionOperands. The ordering does not change when they are nested.

This rule only apply to messages contained in a Combined-Fragment through a InteractionOperand.
Chapter 7

Interaction Model Specification

This chapter gives a description of how the transformation rules from chapter 6 are added to the existing model transformation described in chapter 5. The chapter begins with a short description of new constructs in the UML AST and then an introduction of how UML 2 Sequence Diagram (SD) is related to the nodes in the OML AST. Secondly the key features of the actual transformation from UML SD to VDM traces is given, primarily concerning the changes needed to handle SD in the already existing model. Finally a summary is made discussing what has been specified in the model and which difficulties had been discovered.

7.1 Subset of UML AST in relation to sequence diagrams

This section shortly describes the constructs of the UML AST used to specify constructs of a UML SD. References to a detailed description in Appendix E is supplied for each construct.

The main constructs of the UML AST are shown below:

**Interaction** An interaction represents a single sequence diagram. It holds the following:

- lifelines : set of Lifeline which represents an instance of a class (see section E.2.2).
- fragments : set of InteractionFragment an interaction fragment is a piece of an interaction (see section E.2.3).
- messages : seq of Message a Message defines a particular communication between Lifelines of an Interaction (see section E.2.4).

```
Interaction ::
    name : String
    lifelines : set of LifeLine
    fragments : set of InteractionFragment
```
messages : seq of Message;

Listing 7.1: AST of a Interaction.

**LifeLine** Listing 7.2 shows the type LifeLine, which represents an instance of a class via the optional represents : [Type] (see section E.1.3). If Lifeline do not reference a class it will be ignored during a transformation.

```
LifeLine ::
    name : String
    represents : [Type];
```

Listing 7.2: AST of a LifeLine.

**Message** Listing 7.3 shows the type Message of the Interaction, which includes, but is not limited: sendEvent : Mos references the specification of the sending of the Message and receiveEvent : Mos references the specification of the reception of the Message.

```
Message ::
    name : String
    sendEvent : Mos
    sendReceive : Mos
    ...
```

Listing 7.3: AST of a Message.

### 7.2 Transformation Specification Overview

Transforming a UML SD into a VDM trace definition is carried out according to the rules defined in section 6.2. To illustrate how the rules apply to a SD, a VDM trace statement is shown in Listing 7.4 and an informal presentation of the populated UML AST is shown in Listing 7.5. The two Listings are supplemented with Figure 7.1, which supplies the reader with the link between Listing 7.5 and the SD resulting from the trace statement in Listing 7.4
7.2. TRANSFORMATION SPECIFICATION OVERVIEW

```vdm
class UseStack
  instance variables
  s : Stack := new Stack();
  traces
  testTraceStack = s.Reset();
  s.Push();
end UseStack
```

Listing 7.4: VDM trace statement.

```vdm
NamedTrace
  name = "testTraceStack"
  defs =
    [ SequenceDefinition
      defs =
        [ DefinitionItem
          test =
            MethodApply
              variable_name = "s"
              method_name = "Reset"
        ,
          DefinitionItem
          test =
            MethodApply
              variable_name = "s"
              method_name = "Push"
              regexpr = ZeroOrOne
        ]
    ]
```

Listing 7.5: Populated AST showing the trace from Listing 7.4.

Figure 7.1: Transformation between OML AST and UML SD.

Figure 7.1 shows an example of the link between the OML AST and a UML SD. On the right side of Figure 7.1 the SD is shown. It consists of the lifelines `UseStack` and `s : Stack`. The message `Push` is placed inside an `InteractionOperand`, which has the interaction operator `loop` guarded by the expression `[0; 1].

To the left, the OML AST definitions from Listing 7.5 is shown. Each name from the
Chapter 7. Interaction Model Specification

Definition is linked to the element in the SD that they represent in a trace statement. To illustrate this, the link to a MethodApply expression represents a Message by having the variable name set to the name of the life line at the receive end of the message where the method name is set to the operation that the message represents. (Complies with rule 20 on page 94). A DefinitionItem represents a MethodApply expression. If the method apply expression is inside a CombinedFragment of kind `loop`, the regexpr of the DefinitionItem is set according to the guard of the operand associated with the message. (Complies with rule 23 on page 96).

7.3 Transforming UML SD to VDM Trace

The transformation of a UML Sequence Diagram (SD) into a VDM trace definition is carried out between the UML AST and OML AST. The class `Uml2Vdm` shown in Figure 5.4 in section 5.3, is extended with new functions implementing the rules from chapter 6.2. Apart from the `Uml2Vdm`, an extension has been added to the `Xml2Uml` class, which handles the population of the UML AST. The extension added to `Xml2Uml` is carried out in the same manner as in section 5.5.1.

The transformation process from SD to trace statement can be split up into steps describing the different stages in the transformation, where messages are used as the starting point since they specify the ordering:

1. Extract trace name
2. Find owning class of trace
3. Create definitions from messages. Transformation of the CombinedFragment.
   - SequenceDefinition: If message is inside a CombinedFragment of type `loop`
   - ChoiceDefinition: If a message is inside a CombinedFragment of type `alt`
4. Extract DefinitionItem from message.
   - MethodApply: Defines on which object and which method the message represents.
   - RepeatPattern: Defines the Constraint of the InteractionOperand which the message is included in.

The name of a trace statement is extracted from the name of an interaction as shown in Listing 7.6 by the function `build_trace` (Conforms with rule 19 on page 93). Additional to this the trace definition is built by the function `getTraceDefinition` which recursively handles all messages. Finally a name representing the owning class of the
7.3. TRANSFORMING UML SD TO VDM TRACE

Figure 7.2: Overview of operations and functions involved in the transformation of a UML SD.

trace is extracted. The name is taken from the LifeLine specified by the sendEvent of the Message (Conforms to rule 18 on page 93).

```
private build_trace : IUmlInteraction ->
map String to IOmlTraceDefinitions
build_trace(interaction)==
let name = interaction.getName(),
messages : seq of IUmlMessage = interaction.getMessages()
in
let defs : IOmlTraceDefinition =
getTraceDefinition(messages, interaction.getFragments(),nil)
in
let ownerClass in set
{m.getSendEvent().getCovered().getRepresents()
| m in set elems messages}
in
{let owr : IUmlClassNameType =ownerClass in owr.getName()}
in
|-> new OmlTraceDefinitions{[new OmlNamedTrace(name,defs)]});
```

Listing 7.6: Build a named trace from a interaction diagram.

When the trace definitions are constructed and mapped to a class name they are attached to the body of its owning class. That is done in the construction of classes along with definitions of instance variables, values etc.

According to the goals of rule 21 (sequencing on page 94), rule 22 (choice on page 95) and rule 23 (repeat pattern on page 96) the function getTraceDefinition is introduced. The function iterates over messages from the SD. For each found Message, it is determined if the Message has one of the following states:
• Stand alone Message

• Messages contained in a CombinedFragment of kind loop with an InteractionOperand specifying the repeat pattern.

• Message contained in a CombinedFragment of kind alt where each message related to an InteractionOperand of the same alt should be joined in a choice statement.

Common to all messages is the order which is preserved during a transformation. This also applies to messages sub grouped by a CombinedFragment where the same rule for ordering applies. The only exception is the CombinedFragment of kind alt where the ordering can be ignored, since alt implies a choice behavior. Figure 7.3 shows a state diagram of the three states a Message can end up in, according to its relation to CombinedFragment and their InteractionOperands.

A state diagram is shown in figure 7.3 presenting the different states a Message can have according to its placement in a SD. The state diagram is a graphical representation of the function shown in listing 7.7. If the current message is a standalone message, e.g. it is not enclosed by an InteractionOperand, the message can be converted directly into a DefinitionItem with a MessageApply expression inside. If the current message is enclosed by an InteractionOperand, there are two choices depending on whether the message is enclosed by a CombinedFragment of loop or alt kind. If a Message is processed and no other Messages exist in the same InteractionOperand, the messages found are grouped in a sequence in the same order they occur in the SD, according to rule 21 on page 94. Listing 7.7 shows a snippet of the getTraceDefinition handling the case where a message is contained in a CombinedFragment with the kind loop. This conforms to a repeat pattern of a MessageApply inside a DefinitionItem, according to rule 23 on page 23.
Figure 7.3: State diagram over the function `getTracesDefinition`. 
getTraceDefinition : seq of IUmlMessage *
            set of IUmlInteractionFragment *
            [IUmlInteractionOperand] -> [IOmlTraceDefinition]
getTraceDefinition(msgs,fg,io) ==
    if len msgs > 0 then
        let m = hd msgs,
            rest = if len msgs > 1 then tl msgs else [],
            cfg = ( f | f in set fg & isofclass(IUmlCombinedFragment,f) ),
            op = getOperand(m,cfg)
        in
        ( --No current CF is select by a IO
            if (io = nil and op = nil) or (io = op )
            then -- no operand => no CF
                ...
            else
                if op <> nil and getCfIoKind(fg,op).getValue() =
                    UmlInteractionOperatorKindQuotes'IQLOOP
                then -- CF = Loop
                    let loopDef : IOmlTraceDefinition =
                        getLoopDef(m,op),
                    restDef : [IOmlTraceDefinition] =
                        getTraceDefinition(rest,cfg,op),
                    defs : seq of IOmlTraceDefinition =
                        if restDef <> nil
                        then [loopDef,restDef]
                        else [loopDef],
                    ret : IOmlTraceDefinition =
                        new OmlTraceSequenceDefinition(defs) in ret
                else
                    ...
        ) else nil;

Listing 7.7: Function getTraceDefinition creates a TraceDefinition representing all messages in the SD.

In listing 7.7 the handling of a Message which is inside a CombinedFragment of kind loop is shown. It can be seen at line 6 that the current message m is extracted and the rest presenting the tail list of all messages. In line 9 an InteractionOperand for the current message is looked up. If an InteractionOperand exists and the operand is inside a CombinedFragment of kind loop, the current message m and the rest of the existing messages rest are dispatched to the getLoopDef function at line 20. The getLoopDef shown in listing 7.8 create a DefinitionItem with the required repeat pattern from the operand of the Message. Finally all messages processed by getLoopDef and rest (Is processed by the getTraceDefinition itself) are grouped and returned in a sequence (According to rule 21 on page 94) as shown in line 24-28, listing 7.7. If instead the CombinedFragment of kind alt had been present, the rest of messages would be passed back to the getLoopDef function to be processed accordingly.
getLoopDef : IUmlMessage * [IUmlInteractionOperand] -> IOmlTraceDefinition
getLoopDef(m,io) ==
  new OmlTraceDefinitionItem([],
    getMethodApply(m),
    getRpEx(io));

Listing 7.8: The function getLoopDef creates a TraceDefinitionItem from a Message.

In Listing 7.8 a DefinitionItem is created from a method apply expression through function getMethodApply. The regular expression specifying the repeat pattern is extracted from the constraint associated with the InteractionOperand.

private getMethodApply : IUmlMessage -> IOmlTraceMethodApply
getMethodApply(message) ==
  let methodName : String =
    message.getSendReceive().getEvent().getOperation().getName(),
  variableName : String =
    message.getSendReceive().getCovered().getName(),
  args : seq of IOmlExpression = []
  in
    new OmlTraceMethodApply(variableName,methodName,args);

Listing 7.9: Function getMethodApply creates a OmlMethodApply from a Message.

The creation of a MethodApply from a Message is shown in listing 7.9 (Complies to rule 20 on page 94). The name of a Message is extracted in line 4, listing 7.9. The name is extracted from the Operation which is linked to the Message as shown in line 4. The type of the objects presented in line 4 is as follows:


The repeat pattern, which is applied to a Message if it belongs to a Combined-Fragment of kind loop is constructed from the guard of the InteractionOperand associated with a message. The guard of an InteractionOperand is a constraint which has two properties:

- minint
- maxint

The repeat pattern of a Message can be created from the minint and maxint of a guard attached to an InteractionOperand, according to rule 23 on page 96. Listing 7.10 shows an example of a guard where minint and maxint is represented as LiteralInteger. The listing shown a subset of the getRpEx function. If min=0 and max is undefined the result would be a repeat pattern that equals ZeroOrMore as stated in line 11.
private getRpEx : [IUmlInteractionOperand] -> [IOmlTraceRepeatPattern]
getRpEx(iOperand) ==
  let guard = iOperand.getGuard(),
    min   = if guard.hasMinint() then let tmp : IUmlLiteralInteger =
            guard.getMinint() in tmp.getValue(),
    max   = if guard.hasMaxint() then let tmp : IUmlLiteralInteger =
            guard.getMaxint() in tmp.getValue()
    in if min = 0 and max = nil then
       new OmlTraceZeroOrMore()
    else
       ...

Listing 7.10: A subset of getRpEx showing how a repeat pattern is extracted from a
    guard of an InteractionOperand.

7.3.1 Summary of traces specification

In the above section a description of one of the key features of the SD to trace trans-
formation has been described namely the handling of Messages and their collabora-
tion with CombinedFragments of kind loop. The functions partly described in the
above section are build_trace, getTraceDefinition, getLoopDef, getRpEx and
getMethodApply as a part of figure 7.1. In addition to this the functions getOperand,
getAltDef and getCfloKind exist and have been fully specified as well. Before the
transformation can be enables the UML AST needs to be populated, a complete spec-
ification has been made for this as well. During the process of specifying the XML
Document to UML AST some difficulties has been discovered as described in section
2.7 on page 40.
Chapter 8

Transformation implementation

In this chapter first a discussion how test can be performed both the VDM model and the executable Java source code. It takes into account the fact that tests not only should be done at the implementation level but also at the specification level. Secondarily the process of code generation is presented along with a description of how the transformation tool fits into the Overture project.

8.1 Testing

VDM++ makes it possible to write invariant, pre- and postconditions which must be satisfied in order for the model to be valid. However, such statements are written by developers and are thus prone to errors. Even though it is possible to prove properties about a model, the risk of unforeseen unknowns is still present, e.g. the model is correct but it is not modeling the intended real world scenario or an invariant may be wrong. A way to further mitigate this risk is by executing test-cases against the model and observe the result. The test-cases may be code-generated and executed on the application level to increase confidence in the code-generator.

Two types of testing have been taken into consideration (1) Script testing and (2) Unit testing to be efficient both types should be performed as automated tests. Automated testing enables test to be performed automatically whenever the system changes. Undesired changes can then be identified right after a automatic test have been performed. Additional to the above a third type of test should be mentioned called regression testing which is the type of test VDM traces aims to cover. Regression testing is any type of software testing which seeks to uncover software regressions for instance a automated system could be setup to run a complete test suite each night to cover side effects from previous bug fixes. By using regression testing in an automated manner side effects of bug fixing will be known shortly after correction and the ability to create statistics of number of known errors can be made to indicate the system quality.
8.1.1 Script testing

A test script is a set of instructions that will be performed on the system under test. This can be done manually or automatically, by automating the process using a script or custom written program. A series of tests can be performed on the system under test to unveil undesired behavior. A script could be made to read argument files, test the system with the arguments and compare the outcome of the execution with result files. The outcome of such a test can then later be analyzed to show if bugs were discovered.

![Figure 8.1: Testing a system with script testing.](image)

8.1.2 Unit test

A unit in software testing can be defined as:

A unit is the smallest possible testable software component.

Generally a unit can be considered as a test in a work breakdown structure, a piece that can be compiled separately or a piece that fits on a single page or screen. In relation to VDM a unit can be defined as a:

- function,
- operation or even a
- class.

No matter which type of component is selected as the smallest testable component, unit testing is a vital level of testing. Since unit test is performed on a small component it becomes easier to design, record, perform and analyze test results. If during a test a defect is discovered the relative small size of the component makes it easier to locate and repair the defect in the component [Burnstein&03]. When designing test cases they should be designed in a way which makes them independent upon each parts of the system under test. An important fact is that the test cannot discover all defects in a system, if a part is not tested or if the test case is incomplete. In the case of VDM the ideally way to use using Unit test would be to enable the same test cases to be performed both on the VDM specification level and the Java application level. This will ensure that the specification and the application reacts in the same way. At the Java level a JUnit framework exists which provides a framework for organizing tests including TestCases and TestSuites. A similar framework exist in VDM called
VDMUnit which is a framework compatible with JUnit. By code generating test cases from VDMUnit it becomes possible to execute them as JUnit test cases at the application level which enables the execution of the same test both at specification and application level. This however does not detect defects in the code generator or in the test itself. The construction of test cases could be automated by the use of an argument and result files. Tests could then be created from an argument and a corresponding desired result by inserting the following line in a test case:

\begin{verbatim}
assertTrue( operation identifier ( argument ) = desired result)
\end{verbatim}

8.2 Java code-generator for VDM

The code generation process consists of two steps (1) Generate specification to Java and (2) Inspect, correct and write required stubs.

Generate specification to Java: The code generation are done by using the Java Java Code generator [CGManJavaPP] of VDM Tools [VDMTools]. The VDM specification is generated to Java which result in one class for each class in the specification.

Inspect, correct and write required stubs: By inspection it is clear that all classes not a part of the OML or UML AST e.g Vdm2Uml and Uml2Vdm, need to have an additional package import since the OML AST is used as an external JAR file already code generated and compressed into a JAR as a part of Overture. Due to minor errors in the code generator such as missing return from code generated cases with no others option a return must be inserted neither as return or throw of an exception. Both the org.overturetool.parser.jar file and the VDM.jar file from CSK should be added project buildpath. Then the code needed to invoke the UML transformation is simple, it consists of a main class enabling the program to operate as a self contained program alone and a class handling the invocation of the OML parser and XMI file read/write. Concerning JUnit all VDM test classes can be code generated as well since VDMUnit is compliant with JUnit 1.3.

8.3 Integrating UML in Overture Tool

The Overture tool has the Eclipse platform as its base which means that the UML transformation should be able to easily plug into the eclipse platform. To enable an easy plug-in of the UML transformation eclipse has a plug-in architecture that enables easy
<table>
<thead>
<tr>
<th>Name</th>
<th>AST level</th>
<th>VDM model</th>
<th>Java source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size (kB)</td>
<td>Lines</td>
<td>Size (kB)</td>
</tr>
<tr>
<td>UML.ast</td>
<td>5.262</td>
<td>212</td>
<td>89.767</td>
</tr>
<tr>
<td>Vdm2Uml.tex</td>
<td>21.684</td>
<td>533</td>
<td>55.728</td>
</tr>
<tr>
<td>Vdm2UmlType.tex</td>
<td>5.172</td>
<td>166</td>
<td>13.341</td>
</tr>
<tr>
<td>Uml2XmiEAXml.tex</td>
<td>19.391</td>
<td>622</td>
<td>53.992</td>
</tr>
<tr>
<td>Uml2Vdm.tex</td>
<td>26.416</td>
<td>609</td>
<td>76.173</td>
</tr>
<tr>
<td>Xml2UmlModel.tex</td>
<td>24.005</td>
<td>558</td>
<td>82.828</td>
</tr>
<tr>
<td>Vdm2Uml.tex</td>
<td>4.260</td>
<td>143</td>
<td>13.106</td>
</tr>
<tr>
<td>Oml2Vpp.tex</td>
<td>412</td>
<td>24</td>
<td>1.631</td>
</tr>
<tr>
<td>Oml2VppVisitor.tex</td>
<td>19.636</td>
<td>681</td>
<td>59.055</td>
</tr>
<tr>
<td>external_IO.java</td>
<td></td>
<td></td>
<td>3.653</td>
</tr>
<tr>
<td>MainClass.java</td>
<td></td>
<td></td>
<td>2.787</td>
</tr>
<tr>
<td>Translator.java</td>
<td></td>
<td></td>
<td>2.863</td>
</tr>
<tr>
<td>XmlParser.java</td>
<td></td>
<td></td>
<td>2.947</td>
</tr>
<tr>
<td>ClassExtractor-FromTexFiles.java</td>
<td></td>
<td></td>
<td>1.621</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>210.743</td>
</tr>
</tbody>
</table>

Table 8.1: Measure of model size on AST, VDM and Java level.

<table>
<thead>
<tr>
<th>Name</th>
<th>VDM to Java</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
</tr>
<tr>
<td>UML.ast</td>
<td>320%</td>
</tr>
<tr>
<td>Vdm2Uml.tex</td>
<td>257%</td>
</tr>
<tr>
<td>Vdm2UmlType.tex</td>
<td>257%</td>
</tr>
<tr>
<td>Uml2XmiEAXml.tex</td>
<td>278%</td>
</tr>
<tr>
<td>Uml2Vdm.tex</td>
<td>288%</td>
</tr>
<tr>
<td>Xml2UmlModel.tex</td>
<td>345%</td>
</tr>
<tr>
<td>StdLib.vpp</td>
<td>307%</td>
</tr>
<tr>
<td>Oml2Vpp.tex</td>
<td>395%</td>
</tr>
<tr>
<td>Oml2VppVisitor.tex</td>
<td>300%</td>
</tr>
<tr>
<td>Total</td>
<td>305%</td>
</tr>
</tbody>
</table>

Table 8.2: Percentage increase from VDM to Java.

The integration of different features. All Java based supporting tools in Eclipse are all plug-ins themselves. Plug-ins interact with each other by extension points in the Eclipse framework.
8.3. INTEGRATING UML IN OVERTURE TOOL

8.3.1 Development of the UML Plug-in

All information about a plug-in is stored in a plugin.xml file. Here the name, version, provider, runtime requirements, dependencies, extensions and extension points of the plug-in is placed. The UML transformation plug-in extents the org.eclipse.ui.actionSets where a menu is added to eclipse where it is possible to invoke the transformations (VDM to UML, UML to VDM). The plug-in depends on the JAR files from the Overture project [OvertureTool] containing the Overture parser and the Overture AST implementation found in appendix G.

Additional to the plug-in a feature need to be created such a feature groups plug-ins and exposes then as a feature that the update site can provide.

8.3.2 Deployment of the Plug-in

Eclipse has a well developed plug-in distribution system use on the Internet. Here plug-ins can be download / updated through the Eclipse Software Updates menu. To enable a plug-in for download through Eclipse an update site must be made and published. A update site is a web site where all the features containing the plug-ins are stored along with a description of each plug-in and its dependencies. When the feature is down loaded all the dependencies are resolved and installed before the feature. When installed Eclipse need to restart to load the feature properly before use. To enable the easy integration in Eclipse a Update Site has been created together with the plug-in and feature projects where all dependencies are specified in the plugin.xml files that makes up the plug-in projects. The update site is compiled to Eclipse version 3.4.1 Classic version. The plug-in is located at Eclipse Update Site for VDM-UML transformation [COMUUpdateSite].
Chapter 9

Concluding Remarks

9.1 Achieved Results

The two main goals of this thesis was to investigate the mapping potential between VDM++ and UML and to devise bidirectional transformation rules for each language construct. An important subgoal was to construct a prototype of a tool capable of performing a transformation between VDM++ and UML 2 Class Diagrams, and UML 2 Sequence Diagrams and the new VDM++ traces [Santos08]. The two main goals and associated subgoals were reached at the end of the project with comprehensible support of both VDM++ and UML 2, in addition to a finalized working tool integrated as an Eclipse plug-in in the Overture context.

9.1.1 Learning outcome

To accomplish the goals presented in the introductory section 1.5, thorough analyses of the UML 1, UML 2 and VDM++ notations have been carried out. The analyses resulted in detailed knowledge of the inner workings of both languages, i.e. their respective concrete and abstract syntaxes and their semantics.

The reason for examining UML 1 is that no more than one tool, Rose-VDM++ Link [RoseMan, CSKCORP], exists, which can perform a model transformation similar to the one made in this thesis. The tool is based on UML version 1.1, hence knowledge hereof was required in order to evaluate the quality of the tool in regards to the subset of UML 1 its supports and how well it preserves semantics.

Correspondingly, the reason for examining UML 2 is that the prototype made as part of this thesis work should support the latest development within UML. Thus, it was necessary to know the differences between UML 1 and UML 2 in order to clarify whether any progress has been made on the UML side since Rose-VDM++ Link was created.
The essential part of the model transformation, i.e. the actual mapping from one language to another, takes place at the abstract syntax level of the languages. The abstract syntax representation of VDM was available at the time this thesis work began in the form of an Abstract Syntax Tree (AST). The corresponding AST for UML had to be made during this thesis work. To reduce the manual workload, it was chosen to use the Overture tool ASTGen, which is also used in conjunction with the OML AST, hence the necessity to construct the UML AST in a fashion similar to the OML AST. This required investigation of how the OML AST is structured and how ASTGen works.

UML diagrams are exchanged between tools using an XML-based standard maintained by the Object Management Group (OMG). It was necessary to investigate this standard to determine to which degree UML tools adhere to the standard and how they deviate from it. That is important in order to know which UML tools will be able to import a UML model produced by the prototype made as part of this thesis. It turned out that the standard is not used by the tools as intended, hence the knowledge of the standard was also be utilized to tweak the prototype to satisfy the needs of selected UML tools.

9.1.2 Concrete achievements

This section describes the concrete achievements of the thesis work. It is structured into subsections, each devoted to a particular topic.

Comparison of the Rose-VDM++ Link and our prototype

The aim of the model transformation was to construct a tool with capabilities similar to the tool Rose-VDM++ Link. This thesis work has two important extensions compared to Rose-VDM++ Link:

Introducing UML 2: Rose-VDM++ Link only supports UML version 1.1 released in the late 1990s. The tool made in this thesis supports UML version 2.1.2, which is the latest version at the time of this writing [OMGUMLHomepage]. The tool made in this thesis has the ability to transform the following types, none of which is present in Rose-VDM++ Link:

- template parameters of classes (explained in section 4.12).
- data type definitions in a class definition (explained in section 4.3).
- union types as constrained associations (explained in section 4.5).
- product types as n-ary associations (explained in section 4.6).
- active classes (explained in section 4.9).

The tool also support a smaller subset of constructs to be transformed back to the VDM++ level. An excellent overview table of the supported features is given in Appendix I.
Introducing traces: Recent research have resulted in the introduction of the concept of VDM++ traces [LangManPPT, Santos08], which is compact representations of test-cases for regression testing. This work has also constructed a model transformation between UML 2 Sequence Diagrams and VDM++ traces.

The use of ASTs in the development

The abstract syntax of VDM++ was available via Overture at the time this thesis work began. It was supplied as an AST in which language constructs are specified as VDM-SL types. The UML AST, which describes the UML abstract syntax, is inspired by the OML AST. The drawback of using ASTs is that it is not directly usable and requires external tools in order to interpret and populate them with actual data. The advantage is that constructs that appear in the original source are abstracted away, leaving only the conceptual, essential elements from the syntax. This makes it easier to focus on the content, rather than on the form.

FM implies abstracting away information which is not considered important for achieving core functionality. The model transformation has been specified in VDM++ to abstract away details not directly related to the model transformation and to enable future refinement of the model. Also, the reduced size of a VDM++ model compared to an full-scale implementation yield a more easily maintainable software project. Tables 9.1, reproduced and adjusted from section , show an excerpt of the key figures regarding the advantages of code-generation and the resulting sub-totals. Table 9.1 show that the use of ASTGen to generate VDM++ classes from the UML AST resulted in no less than 3556 lines of VDM++ model being generated, corresponding to a 1677% difference between the UML AST (212 lines) and the generated VDM++ classes (3556 lines). In addition, the entire model transformation consist of a total of 583 VDM++ classes. The OML AST are responsible for 448 of these, 135 as the result of the UML AST. The figures in Table 9.1 show the difference between the VDM++ model and corresponding code-generated Java-classes. Notice the average reduction of approximately two thirds on manual-work when using code-generation.

<table>
<thead>
<tr>
<th>Name</th>
<th>AST level</th>
<th>VDM model</th>
<th>Java source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Lines</td>
<td>Size</td>
</tr>
<tr>
<td>UML.ast</td>
<td>5.262</td>
<td>212</td>
<td>89.767</td>
</tr>
<tr>
<td>Vdm2Uml</td>
<td>21.684</td>
<td>533</td>
<td>55.728</td>
</tr>
<tr>
<td>Uml2Vdm</td>
<td>26.416</td>
<td>609</td>
<td>76.173</td>
</tr>
<tr>
<td>Om12VppVisitor</td>
<td>19.636</td>
<td>681</td>
<td>59.055</td>
</tr>
<tr>
<td>Subtotal</td>
<td>157.503</td>
<td>5.379</td>
<td>478.725</td>
</tr>
</tbody>
</table>

Table 9.1: Measure of model size on AST, VDM and Java level. Sizes are given in bytes.

If a transformation had been attempted on the data-layer level of UML diagrams, i.e. the raw XML, a risk exists that only a single or few UML tools would be supported due
to the discovery of several violations of the standard for diagram exchange, advocated by OMG. While this is the case for the current implementation (only Enterprise Architect is supported), the UML AST makes it possible to extend the transformation specification according to the UML specification and implement any special requirements from different UML tools in order to support them.

The combination of using ASTs to capture the abstract language syntax and the formal specification of the model transformation has yielded an extendible and flexible solution. The extensible nature of the AST has proven to be the correct solution to pursue, due to the quick and easy extension with Sequence Diagram types in the second phase of the project. In addition, the reuse of the Overture tool ASTGen saved a lot of manual work writing VDM classes representing the different UML constructs.

**Utilized Overture Tools**

Two tools, both part of the Overture project, and several utility-tools, made as part this thesis work, have been utilized to enable going from the data-layer to the abstract syntax and back again.

The Overture tools used are:

**ASTGen:** The abstract syntax of VDM++ was made as part of the Overture project and supplied as VDM-SL types in an AST. The Overture tool ASTGen takes as input an AST and produces as output one VDM++ class and one corresponding Java interface for each type found in the AST. The tool has been utilized to generate VDM++ classes for the UML AST.

**OML Parser:** The tool parses VDM++ classes and populates the OML AST.

**Utility-tools made in this thesis**

The utility-tools made during this thesis work comprise:

**XML Parser:** UML models are represented in an XML structure. The tool parses the XML structure and produces both VDM++ and Java compliant output, making it possible to perform a model transformation on the specification level without code-generating the entire model to Java first.

**XML Serializer:** The intermediate data structure representing a UML model during the model transformation must be serialized to correctly formatted XML in order for UML tools to import it.

**Abstract to concrete syntax deparser:** The abstract syntax of a VDM++ model is iterated through by visiting each construct. The tool produces a VDM++ class as its output, i.e. a notation that complies by the concrete syntax of VDM++. 
9.2. FUTURE WORK

Extending VDMEditor: The VDM++ specification of the model transformation was written using Eclipse. A plugin for Eclipse, called VDMEditor, provides a limited outline of the model, which presented operations and functions of a file with a single VDM++ class in it. VDMEditor was extended with the capability of surveying a multitude of VDM++ classes within a single file by coloring all keywords of VDM++.

VDMTools plug-in for Maven: The work of this thesis is to be included in the Overture product family. A plugin for Maven have been made which enable Maven to execute the type-checker of VDMTools on the model and automatically code-generate Java files when the specification has changed.

Eclipse plug-in and update site: A plugin for Eclipse and a corresponding update site has been made, which enables everyone to download, install and run the model transformation seamlessly.

Testing

The model has also been subject to testing, which increases confidence in the correctness of the model. The tests are specified in VDM++ and can be run against the model itself or be code-generated and run against the code-generated model. This help increase confidence in the model and also in the VDM++ code-generator, i.e. the same tests are performed on both specification and application level.

Bug-reports

A noticeable by-product of the formal specification of the transformation is the numerous bugs detected in VDMTools and VDMJ\(^1\), which were reported to CSK and Nick Battle at Fujitsu, respectively [Fujitsu] [CSKCORP]. Both CSK and Fujitsu have been kind enough to correct the bugs as they surfaced, leading to increased quality of both products.

9.2 Future Work

To reach a higher level of completeness, the transformation tool requires attention in the areas presented in this section.

The transformation from UML Class Diagrams to VDM++ is not complete. The overview table in Appendix I shows the extent of the transformation. The rules describing the transformation are complete, but they have not been formally specified.

Additionally, if a user maintains both a VDM++ and a UML representation of a software system, and alterations are made to both models, it makes sense to provide

\(^1\)Type-checker and interpreter, developed as part of the Overture project. Used for continuous testing to further refine the model.
a facility enabling a merge of two such models. This feature is present in the Rose-VDM++ Link. However, the merge of UML and VDM++ models is described only in theory in this thesis work, because it was considered of less importance from an academic point of view.

The map between VDM++ data types and UML nested classes should be further investigated. The transformation tool only has the notion of primitive types, e.g. int and char, hence data types defined explicitly in a VDM++ class definition do not map as the corresponding UML meta-class DataType. Instead, the VDM++ data types are treated as UML inner classes. That approach should be subject of a discussion with a larger part of the VDM community.

The transformation from UML Sequence Diagrams to VDM++ traces has been completed. A limitation is the let and let be st statements, which require further attention to fully support a round-trip between VDM and UML Sequence Diagrams. The other direction, from traces to Sequence Diagrams, remains as future work. However, the transformation of traces to Sequence Diagrams should be a trivial task since all rules support this already.

Further investigation of additional UML diagram types could also be carried out to determine whether other diagram types may be of interest. For example, it would be interesting to visualize VDM++ class behavior using a UML State Diagram.

Concerning the VICE extension of VDM++, a range of unclarified possibilities exist, among which the ability to use UML Deployment Diagrams to show distribution of classes on different CPUs, and the use of UML Sequence Diagram features like timing constraints and par\textsuperscript{2} to show time constraints and parallelism [UMLSuperstructure2.1.2, p538]. In this regard, the use of OMG Systems Modeling Language (SysML) could be of interest. SysML reuses a subset of UML 2 and provides additional extensions to satisfy the requirements of the language. SysML is designed to provide simple but powerful constructs for modeling a wide range of systems engineering problems, i.e. problems related to the abovementioned [SysML].

### 9.3 Overall Conclusion

The initial discussion regarding UML and VDM++ concerned how VDM++ could benefit from tapping into the world of visual modeling using UML. The rationale for choosing UML 2 Class Diagrams was mainly due to the fact that Rose-VDM++ Link already supported them, albeit only version 1.1, hence developers already used UML Class Diagrams to a certain extent. The recently introduced concept of traces [Santos08] opened up the possibility of also connecting UML Sequence Diagrams and VDM++. The rationale for choosing sequence diagrams and traces for the second phase was the news-value it could generate around an already new concept. In addition, the combination of Sequence Diagrams and VDM++ traces should be viewed as a proof-of-concept for

\footnote{Denoted that the behavior of multiple operands is being executed C.3.}
observing if the combination increase the understanding of traces in model development and testing.

In order to determine the differences between UML 1 and UML 2, it was necessary to study both specifications closely, i.e. the Superstructure and Infrastructure specifications of UML 1.4.2 [UML1.4.2] and UML 2 [UMLInfrastructure2.1.2] [UMLSuperstructure2.1.2]. Moreover, the VDM++ language specification was thoroughly examined to make sure the semantics of VDM++ was fully understood. Rose-VDM++ Link is the only tool available on the market which is able to perform a round-trip between VDM++ and UML. The mapping specification for Rose-VDM++ Link was also meticulously examined in order to understand the capabilities of the tool.

To summarize, both main goals have been reached, as described in the following.

VDM++ and UML Class Diagrams

- **VDM++ to UML Class Diagram**: Transformation rules for all relevant VDM++ constructs have been specified and the constructs can be mapped to a UML 2 Class Diagram, including template parameters, which is only supported by Overture. The excluded construct comprise invariants, pre- and postconditions, which were considered irrelevant to the purpose of a visual modeling language such as UML. Also, data types are not mapped as the UML metaclass `DataType` although rules are specified to aid the mapping. The reason for this was prioritizing of tasks in a tight time-frame.

- **UML Class Diagram to VDM++**: Transformation rules are specified as in the VDM++ to UML Class Diagram mapping.

UML Sequence Diagrams and VDM traces:

- **UML Sequence Diagram to VDM++ traces**: Transformation rules for all relevant constructs, except `let` and `let be st`, have been specified. The fragments `loop` and `alt` of a Sequence Diagram, used to express procedural logic, map to corresponding VDM++ repeat-patterns of traces.

- **VDM++ traces to Sequence Diagram**: Due to time constraint no final mapping is completed from VDM++ traces to UML Sequence Diagrams. However, transformation rules stating how to transform VDM++ traces constructs to Sequence Diagram constructs have been as rules in a natural language.

In order to deliver an executable prototype a number of utility-tools have been made, e.g. VDMTools plug-in for Maven, Eclipse plug-in, Eclipse Update site, Outline extension of VDMEditor etc.

The transformation tool has been developed to such an extend, that it has been used to produce most of the diagrams in this thesis. Even though the transformation is not complete, it should be seen as a solid starting point both for the completion of the transformation itself and for further development of the connection between VDM++ and.
UML modeling. In particular, bidirectional transformation rules have been formulated for all relevant language constructs and the most complicated rules have been specified using VDM++ and are part of the prototype. The remaining work of incorporating the outstanding rules is considered fairly trivial, hence the academic value is limited. However, it is definitely possible to extend the thesis work presented here, but to do so will open up entire new research areas.

This thesis work has minimized the distance between VDM and UML with regards to the model transformation and opened up the possibility for others to take it further. We sincerely hope, that the tool will enrich the Overture tool-set and that the project will succeed in making VDM++ accessible even to developers not previously familiar with formal methods.
References


Sergiu Dascalu Peter Hitchcock. An Approach To Integrating Semi-formal and Formal Notations in Software Specification. Technical Report, Faculty of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, NS, B3H IW5, Canada, 1 (902) 494 6449, 2002. [cited at p. 9, 10, 14, 15]


9.3. OVERALL CONCLUSION

[cited at p. 10, 11, 43, 45, 46]


[Fujitsu] Nick Battle at Fujitsu Services is responsible for the development of VDMJ. http://uk.fujitsu.com/. [cited at p. 11, 119]


[IFAD] IFAD A/S developed and maintained VDMTools until 2004. after which the intellectual property rights for VDMTools were acquired by CSK Holdings Corporation, Japan. [cited at p. 44]


rCOS - Refinement of Component and Object Systems
http://demo.iist.unu.edu/rcos/.

CSK Holding Corporation. VDMTools: The Rose-VDM++ Link, ver.1.1.

CHAPTER 9. CONCLUDING REMARKS


CHAPTER 9. CONCLUDING REMARKS

32]


Appendices
Appendix A

Overture Workshop 5 in Braga Portugal

A.1 Participation in Workshop

The Overture core team arrange a workshop at the University of Minho in Braga, Portugal, 8th and 9th of November 2008. The overall topic of this workshop was focus on Tool Development. The Overture core team invited us to participate and give a presentation of our M.Sc. thesis project.

The goal of the workshop were to develop a wider knowledge base of tool support for VDM and exercise different aspects of developing software of the Overture open source platform on top of Eclipse. Aspects covered included:

- Kernel functionality developed on top of the abstract syntax using VDM++ and with code generation to java.
- User interface functionality focusing on how to develop Eclipse plug-ins.
- Testing using VDMUnit and JUnit of Overture components.

A.2 What did we gain from the Workshop

The workshop was planed in the final phase of our project which meant that we in advance had worked with the VDMTools and a subset of the Overture components: Overture Parser, the Overture OML AST, ASTGen and VDMJ. This meant that we before the workshop had a good insight in which tools existed. Participating in the workshop we got introduced to other tools like Byaccj, jflex, the new Version of VDMUnit and Maven project management. Both the VDMUnit and Maven did integrate nicely with the UML transformation project. After the workshop we decided to enrich the VDM
- Maven integration by implementing a VDMToolMaven plug-in to enable type check directly in Eclipse followed to the invocation of the code generator for Java if changes had been made at the specification level. The implementation of the automatic code generator resulted in multiple upgrade requests to CSK concerning the ability to code generate single classes and better specification of packages.

A.3 Workshop conclusion

The aim of the workshop were to spread knowledge of the Overture project, try out the eclipse plug-in platform and introduce the VDMUnit framework. As a result of the workshop we gained a lot of knowledge of the Overture project and which tool currently exist among JFlex and Byaccj, as a part of Overture. We got the UML transformation rearranged into the desired Maven structure and checked in at the repository at SourceForge located at: Overture source at SF. Additional to the rearrange of Java code plus VDM specification we changed all the VDM tests from the old test framework into the new VDMUnit which supports JUnit 1.3. To raise the Maven structure from the Java level to the specification level we started development on a VDMTools plug-in for Maven with the ability to type check and auto code generate.

---

1Source Forge Overture tool project https://overture.svn.sourceforge.net/svnroot/overture
Appendix B

Omitted UML 1 Constructs

The UML 1 constructs presented in this appendix have been omitted in this thesis for either three reasons: first, some exist at a higher level of abstraction and thus leave certain design decisions open. A tool is not capable of making those design decisions when performing a model transformation. Second, the concepts represented by some constructs cannot be implemented in VDM, hence the construct has no semantical counterpart in VDM. Or third, the elements are not considered relevant from an academic point of view.

B.1 Association

VDM does not have the notion of a shared attribute by value. Every object is referred to via the Object Reference Type [Fitzgerald&05, p78].

**Composite aggregate association:** The composite aggregate represents a whole/part relationship, i.e. it is a strong form of aggregation which means that the composite object has sole responsibility for the creation and destruction of the instances it owns. An object may be part of at most one composition at a time, thus forming a directed acyclic graph (DAG) of composite objects. Destruction of a higher level object will cause a cascading delete on the DAG as an direct implication of the one-owner property [UML1.4.2, p86].

**Shared aggregate association:** A shared aggregate exists semantically between an ordinary association and a composite association. The shared object may be owned by several aggregations and it may shift owner over time [UML1.4.2, p86]. The shared aggregate is tricky because it is not tightly defined [UML1.4.2, p86]. It is physically impossible for two distinct instances of a class to own the same instance of another class by value. So what exactly does a modeler mean by a shared aggregation? As Jim Rumbaugh says, "Think of it as a modeling placebo"
APPENDIX B. OMITTED UML 1 CONSTRUCTS

[UMLDistilled, p67] and so this thesis exclude the shared aggregate association due to its undefined nature.

B.2 Dependency

A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements. Dependencies cannot be transformed to VDM, because it denotes a dependency across levels of abstraction. For example, if two packages are dependent on each other, not at model level, but at implementation level, at thus have a dependency between them, that cannot be represented in VDM. [UMLSuperstructure2.1.2, p78].

B.3 Derived Element

A derived element is an element, whose value can be directly computed from values of its enclosing classifier. For example, if a class has an enumeration gender and a boolean attribute isMale, then either the former or the latter is a derived element [UMLSuperstructure2.1.2, 32]. The derived element has been omitted, because a machine in incapable of deciding whether the value of an element is directly computable from other values. Such a capability require knowledge of the semantics of the elements in some form of meta-data associated with the elements.

B.4 Package and subsystem

A package is a generic ordering of classifiers [UMLSuperstructure2.1.2, 123]. The ordering is identified by a name, which is the namespace of the package. In UML, packages may also have visibility. The concept of a package and package visibility is not part of VDM.

B.5 Association Class

An association class denotes an association with class-like properties such as state and operations. The semantics of such an association is a combination of the semantics of an ordinary association and of a class [UML1.4.2, p87]. An association class adds an extra constraint, in that there can be only one instance of the association class between any two participating objects. An example is given in Figure B.1. The primary use of an association class is to emphasize the existence of one instance of a class due to an association between instances of two other classes. Figure B.2 shows an example of a design where such an emphasis is practicable.

In figure B.1 an association class can be utilized to denote information specific to the association itself but not to either two classes, e.g. as it is done in the upper diagram.
When an instance of a Professor class is associated with an instance of a Student class, there will also be an instance of a ThesisProject class. The relationship could also be modeled as in the lower diagram. There is a subtle difference between the two diagrams. In the upper diagram, the case of a thesis written by more than one Student will cause duplicate instances of ThesisProject to exist. That is not the case in the lower diagram where a single ThesisProject may be associated to two Students, thus the correct design decision here is to not use an association class.

In figure B.2 a the classes Person and Skill with association class Competence as the association between them. When an association between an instance of Person and Skill exist an instance of Competence also exist. Where could Competence go if it were an ordinary class? Placing it between Person and Skill is meaningless since a Person is expected to first have a skill and then a level of competence in exercising that skill (middle diagram). The main difference between the topmost and bottom diagram is that the bottom suggests that only Skill knows about Competence. That decision is left open in the topmost diagram. Common to both examples is the fact, that association classes leaves the design-decision of the exact implementation of the association class open. Thus it is not directly usable in the context of this thesis.

B.6 Interface

An abstract class with no implementation is semantically the same as an interface. An interface specifies the externally visible operations by a classifier that implements it. An interface contains no attributes or outgoing connections (i.e. an interface may be the target of a one-way connection). VDM do not have the concept of interfaces, only of
abstract classes, which suffice due to the opening sentence of this section.

### B.7 Realization

Realization is best described by relating it to generalization: Generalization connects a subclass to a superclass, hence it is another name for inheritance. Realization connects a class to an interface, hence it is another name for implementation. Since interfaces do not exist in VDM, realization is of no interest in this thesis.

### B.8 Attributes of metaclass Class

A classifier has a number of attributes that dictates the role of the classifier. Common to both regarding VDM is that VDM do not pose any restrictions on what modelers are allowed to do with a VDM class.

- **isRoot**: Specifies whether the classifier may inherit from other classifiers or not. A value of `true` states a lowest level superclass in a class hierarchy.

- **isLeaf**: Specifies whether the classifier may be inherited or not. A value of `true` states that derivation from the class is illegal.

- **visibility**: Denotes the visibility of a class. However, VDM classes are always public.
B.9 Concurrency

The enumeration CallConcurrencyKind defines three types of concurrent behavior a class may utilize [UML1.4.2, p102]. The different types of behavior can be modeled in VDM using history counters [Fitzgerald&05, p280] but have been omitted due to lack of visual benefit.

**sequential:** Callers must coordinate so that only one call to an Instance (on any sequential Operation) may be outstanding at once.

**guarded:** Multiple calls from concurrent threads may occur simultaneously to one Instance (on any guarded Operation), but only one is allowed to commence.

**concurrent:** Multiple calls from concurrent threads may occur simultaneously to one instance (on any concurrent Operation). All of them may proceed concurrently with correct semantics.

B.10 DataType

The meta-class DataType represents the general notion of being a data type (i.e., a type whose instances are identified only by their value). A primitive type is a data type implemented by the underlying infrastructure and made available for modeling. Typical use of data types would be to represent programming language primitive types, e.g. int, char, etc. Data type definitions in a VDM class definition are not mapped as the meta-class DataType. They are instead mapped as UML inner classes, because they resemble an inner class more than a simple data type.
Appendix C

Omitted UML 2 Constructs

The UML 2 constructs presented in this appendix have been omitted in this thesis for either three reasons: first, some exist at a higher level of abstraction and thus leave certain design decisions open. A tool is not capable of making those design decisions when performing a model transformation. Second, the concepts represented by some constructs cannot be implemented in VDM, hence the construct has no semantical counterpart in VDM. Or third, the elements are not considered relevant from an academic point of view.

C.1 Internal Structure of a Class

Specifies that certain classes are parts of another class, i.e. the parts do not have the same semantic meaning if found outside the scope of the owning class. This is an abstraction not applicable to VDM because it involves unmade design decisions that cannot be resolved by a machine [UMLSuperstructure2.1.2, p201].

C.2 Message kind

The meta-class MessageKind is an enumeration of the following values:

- **complete**: Origin and target of message is known.
- **lost**: Origin of message is known, but target is unknown.
- **found**: Origin of message is unknown, but target is known.
- **asynchronous signal**: Designates creation of another lifeline object.
- **delete**: Designating the termination of another lifeline.
- **reply**: Designating a reply message to an operation call.
The enumeration has no practical use in relation to VDM and UML SD, because only VDM traces are considered, thus any potential use of the semantics no longer apply.

### C.2.1 Part decomposition

A message that points back to its originating object can represent a recursive message or a message calling another method on the same object. A more decorative way to show more than one lifeline stemming from an object, is to use part decomposition [UMLSuperstructure2.1.2, p513]. Figure C.2 shows an object of a class with an important internal structure. Part Decomposition can be utilized to describe the behavior of that internal structure.

Figure C.2: Port1 and Port2 represents the UML 2 concept Port, which specifies a distinct interaction point on an object lifeline [UMLSuperstructure2.1.2, p196]
C.3 Fragment

The following fragments are not applicable in the context of this thesis. They are abstractions useful to humans but not to a model transformation.

**Negative (neg):** Denotes an invalid series of messages.

**Ignore (ignore):** Indicates that one or more message types are of no interest even if they occur. These message types can be considered insignificant and are implicitly ignored if they appear in a corresponding execution, i.e. the execution context modeled by the Sequence Diagram will ignore the message types because they are of no interest to the current execution, however, they may be significant in another context [UMLSuperstructure2.1.2, p489].

**Consider (consider):** A consider operator is in effect the opposite of the ignore operator, i.e. any message not present in the consider operator should be ignored.

**Break (break):** A break represents a behavior that is performed instead of the remainder of the enclosing fragment, thus it can be thought of as a way to model exception handling. If the guard of the break evaluates to true the break operand is executed and the rest of the enclosing sequence is ignored. If there is no guard the choice between a break and the remainder of the fragment is non-deterministic.

**Parallel (par):** Denotes that the behavior of multiple operands is being executed in parallel, i.e. each operand in the fragment represents a thread of execution done in parallel. The contents of each operand may be interleaved arbitrarily as long as the ordering imposed by each operand as such is respected. A notation shorthand for parallel combined fragments is Coregion, as showed in Figure C.3.

![Coregion notation](image)

**Figure C.3:** The method calls $m_2$ and $m_3$ may appear in any order [UMLSuperstructure2.1.2, p522].

**Weak Sequencing (seq):** Denotes an sequence of messages that can occur on one or more lifelines. There is no decided order of messages within a seq that do not
share the same lifeline. The only requirement is that all the events in a preceding operand must be completed before the following operand can start. Weak sequencing may be decorated with Continuations (see Alternative (alt)).

**Option (opt):** Denotes a binary choice of behavior where either nothing happens or the operand happens. It is equivalent to an if-then construct.

**Strict Sequencing (strict):** Encloses a series of messages which must be processed in the given order. It is a stronger notion of weak sequencing. Strict sequencing may be decorated with Continuations (see Alternative (alt)).

**Critical Region (critical):** A critical section declares an atomic subsequence which cannot be interleaved by any participant within the enclosure. Other operands (e.g. par) may imply the possibility to interleave a critical region, but this is prevented.

![Diagram](image)

Figure C.4: Example of a parallel and critical region to denote concurrent emergency calls and focus on forwarding 911 calls [UMLSuperstructure2.1.2, p486]

**Assertion (assert):** Denotes an assertion of the messages within the enclosure. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace.

### C.3.1 ConsiderIgnore Fragment

A ConsiderIgnoreFragment is a kind of combined fragment with an interaction operator of kind Ignore/Consider [UMLSuperstructure2.1.2, p489]. The interaction
The operator `ignore` designates that there are some message types that are not shown within this combined fragment. These message types can be considered insignificant and are implicitly ignored if they appear in a corresponding execution. Alternatively, one can understand `ignore` to mean that the message types that are ignored can appear anywhere in the traces. Conversely, the interaction operator `consider` designates which messages should be considered within this combined fragment. The concept cannot be used in this thesis, because marking particular messages as more or less significant than other messages has no meaning in relation to VDM traces.

### C.3.2 InteractionUse

If a CombinedFragment is used in multiple Sequence Diagrams, it may be represented more clearly as an InteractionUse. An InteractionUse is a form of z-layering or zooming capability which allows the modeler to refer to a CombinedFragment instead of drawing it repeatedly. The InteractionUse is shown as a CombinedFragment symbol where the operator is called `ref`. 
Appendix D

Significant changes to the UML meta-model

This chapter contains a description of the changes of the UML meta-model from UML 1 to UML 2. The changes do not have a direct impact for the user of UML, because the changes are on the abstract syntax part of the UML meta-model. However, they are interesting in order to better understand some of the design decisions made by UML tool vendors (e.g. the use of Collaboration by EA in UML 2 SDs, see section 2.6.2 and appendix E.2.1 for further information).

D.1 Deprecated UML 1 meta-classes

**ScopeKind:** The meta-class has been replaced with the meta-attribute `isStatic`. `ScopeKind` was an enumeration of the values `{instance}` and `{classifier}` and was used by meta-classes Feature and AssociationEnd, which now use `isStatic` instead.

**AssociationEnd:** The meta-class has been demoted to the attribute `memberEnd` of Association [UMLSuperstructure2.1.2, p62].

**Multiplicity:** The meta-class has been replaced with the abstract meta-class `Multiplicity-Element`, which other meta-classes inherit in order to have a multiplicity. In UML 1, various meta-classes had attributes of type `Multiplicity` in order to specify multiplicity. In UML 2, that attribute is supplied by inheritance instead [UMLSuperstructure2.1.2, p110].

**Collaboration:** In UML 2, the meta-class Collaboration has been revoked and Interaction has been promoted to encompass in general any kind of interaction [UMLSuperstructure2.1.2, p501]. It is worth noticing that Interaction is a
subclass of Classifier, which is also the superclass of Class from CD. Hence, any of the two meta-classes may be put instead of Classifier.

**ClassifierRole:** Participants of Interaction is modeled by Lifelines instead of ClassifierRoles [UMLSuperstructure2.1.2, p499].

**Sequencing of messages:** The sequencing of messages by sequence numbers as used in UML 1 SD has been moved to Communication Diagrams, which correspond to simple UML 2 SDs. CombinedFragment is used to indicate sequencing in UML 2 SD.

### D.2 New UML 2 meta-classes

**ValueSpecification:** The meta-class has been added in UML 2. A ValueSpecification is used to identify a value or values in a model. The range of a ValueSpecification may be restricted by Constraint, which specifies additional semantics for one or more elements [UMLSuperstructure2.1.2, p74]. The use of ValueSpecification is a further refinement of the UML meta-model.

**isUnique:** For a multi-valued multiplicity, this meta-attribute specifies whether the values in an instantiation of the element are unique, i.e. whether it is possible to have several links associating the same set of instances [UMLSuperstructure2.1.2, p110].
Appendix E

Specification of the UML Abstract Representation

This appendix describes how the UML AST corresponds to the UML Superstructure Specification [UMLSuperstructure2.1.2]. A future description on how the AST is comprised and a description on tool support for creating a AST to use in VDM is described in section 5.1 where a walk through is done with a small example showing the construction and how to apply the ASTGen\(^1\) tool to the AST to create VDM specification classes that can be used in the modeling process.

The Superstructure defines the notational representation of the concrete meta-classes and how the various classes are interconnected. Some of the concrete classes do not have a graphical notation and serve only as “glue”, e.g. the Mos used in SDs which link Messages to Lifelines. Both graphical and non-graphical constructs are equally important in order to allow further extensions to the UML AST while maintaining compliance with the UML specification. Some constructs, however, have been omitted from the AST of two reasons:

1. They did not add value to the UML AST in the context of this thesis, i.e. some constructs exist to bind together parts of the UML specification,

2. and meta-classes that makes use of multiple inheritance are collapsed into the inheriting class, because multiple inheritance is not possible in Java. Concerning the latter case it relates only to abstract classes in the meta model where the concrete class is used instead of its abstract super class.

All UML AST constructs have a UML counterpart by the same name, unless stated otherwise. The UML meta-classes corresponding to types defined in the UML AST can

\(^1\)Tool to convert VDM-SL type definitions to VDM classes etc. see section 5.1.1.
be found in section E.3. This approach reduces the massive amount of citations that would otherwise have been present in the following text.

E.1 Class Diagram

E.1.1 Model and ModelElement

The top-level element of the AST is a Model which comprises a name and a set of ModelElement. UML does not have the notion of a diagram that contains a number of elements. Rather, a collection of elements constitute a diagram conceptually, hence the AST Model and ModelElement do not have a counterpart in UML.

The ModelElement shown in Listing E.1 consist of the following:

Class, Association and Constraint, which are described in sections E.1.2, E.1.4 and E.1.5 respectively.

Collaboration: The semantics of Collaboration has changed since UML 1. However, Collaboration is used here in a UML 1 context, because the primary UML tool used in this thesis uses it. See section E.2.1 for further explanation.

---

Model ::
 name : String
 definitions : set of ModelElement;
 ModelElement = Class | Association | Constraint | Collaboration;

Listing E.1: Model and ModelElement.

E.1.2 Class

The type Class is shown in Listing E.2. It contains a number of types which are described in this subsection.

---

Class ::
 name : String
 classBody : set of DefinitionBlock
 isAbstract : bool
 superClass : seq of ClassNameType
 visibility : VisibilityKind
 isStatic : bool
 isActive : bool
 templatesignature : [TemplateSignature];
DefinitionBlock

The type Class defines classBody as a set of DefinitionBlock, shown in Listing E.3, which has no UML equivalent and act as a container for the following three types shown in Listings E.4, E.5 and E.6, respectively:

OwnedOperations: Defines a set of Operation.

• An Operation has a MultiplicityElement that defines the multiplicity of the return parameter, e.g. two or more to denote a collection as return parameter. Notice, that the multiplicity of the return parameter of Operation is actually defined as the attributes lower and upper in the UML specification. However, MultiplicityElement already define those two attributes, hence the reason they have been replaced by MultiplicityElement in this thesis work.

• The optional Parameters denote the parameters of the Operation. The direction of the Parameters are given by the enumeration ParameterDirectionKind.

OwnedProperties: Defines a set of Property. A Property represents a declared state of one or more instances in terms of a named relationship to a value or values, e.g. a named attribute of a class.

NestedClassifiers: Defines a set of Type which enables types e.g. ClassNameType to be nested inside other classes as a result of this classes can be nested.

DefinitionBlock =
    OwnedOperations | OwnedProperties | NestedClassifiers;

Listing E.3: DefinitionBlock.

OwnedOperations ::
    operationList : set of Operation;

Operation ::
    name : String
    visibility : VisibilityKind
    multiplicity : MultiplicityElement --aka return type
isQuery : bool

--aka return type

type : [Type]

isStatic : bool;


MultiplicityElement::

isOrdered : bool

isUnique : bool

lower : nat

upper : [nat];

Listing E.5: OwnedProperties and Property.

OwnedProperties ::

propetyList : set of Property;

Property ::

name : String

visibility : VisibilityKind

multiplicity : [MultiplicityElement]

type : Type

isReadOnly : [bool]

default : [ValueSpecification]

isComposite : bool

isDerived : [bool]

isStatic : [bool]

ownerClass : String

qualifier : [Type];

Listing E.6: NestedClassifiers.

NestedClassifiers ::

typeList : set of Type;

ClassNameType

See section E.1.3
VisibilityKind

The element Package of the enumeration visibility : VisibilityKind from Class has been omitted, because the notion of packages do not exist in VDM.

\[
\text{VisibilityKind} = \langle \text{PUBLIC} \rangle \mid \langle \text{PRIVATE} \rangle \mid \langle \text{PROTECTED} \rangle ;
\]

Listing E.7: VisibilityKind.

TemplateSignature and TemplateParameter

The optional templatesignature : TemplateSignature of Class bundles the set of TemplateParameter, for the templated class.

\[
\text{TemplateSignature} ::=
\text{templateParameters} : \text{set of TemplateParameter};
\]
\[
\text{TemplateParameter} ::=
\text{name} : \text{String};
\]

Listing E.8: TemplateSignature and TemplateParameter.

E.1.3 Type

Type has one constituent which is not self-explanatory:

\[\text{superClass} : \text{ClassNameType} : \text{Defines Class}
\]

introduced in this thesis to ease the way a class is referenced. In the UML superstructure a class is referenced as an object which means that a class that should be referenced should be fully constructed with all its members such as properties, operations etc. By introducing ClassNameType this can be avoided because class names in VDM are considered unique, this means that the use of ClassNameType does not compromise the UML structure it is just a weakening of the reference. ClassNameType do not have a UML counterpart.

\[
\text{Type} = \text{BoolType} \mid \text{IntegerType} \mid \text{StringType} \mid \text{UnlimitedNatural} \mid \text{VoidType} \mid
\]
APPENDIX E. SPECIFICATION OF THE UML ABSTRACT REPRESENTATION

CharType |
ClassNameType;

Listing E.9: Type.

ClassNameType ::
  name : String;

Listing E.10: ClassNameType.

E.1.4 Association

The Association of ModelElement is shown in Listing E.11. Association defines id:Id which is used to associate an Association with a Constraint (see section E.1.5). Notice, that the UML specification states the type of the UML attribute constrainedElements as Element, hence the UML AST construct Id should be replaced with a new UML AST construct Element in order to conform to the UML specification. However, multiple inheritance will then be introduced to the UML AST because Element is a top-level superclass, hence for now the Id must suffice. Alterations may be made in the future to accommodate for the missing UML AST Element construct. Listing E.12 shows how Id is defined.

Association ::
  ownedEnds : set of Property
  ownedNavigableEnds : set of Property
  name : [String]
  id : Id;


String = seq of char;
Id = String

Listing E.12: String and Id.

E.1.5 Constraint and ValueSpecification

The type Constraint, shown in Listing E.13 defines a set of Id, which is the set of elements being constrained by the constraint. The specification of the constraint is defined
by specification: ValueSpecification of Constraint and it contains the following:


---

Constraint ::
constraintElements: set of Id
specification: ValueSpecification;

ValueSpecification = LiteralString | LiteralInteger;


---

LiteralString ::
value: String;

LiteralInteger ::
value: nat;

Listing E.14: LiteralString and LiteralInteger.

---

E.2 Sequence Diagram

The description so far has covered the abstract syntax of UML Static Structure diagram. The last entry in the ModelElement in the beginning if the UML AST is Collaboration, which represents a UML SD.

E.2.1 Collaboration

As mentioned in section E.1.1, Collaboration is used in this thesis with its UML 1 semantics, hence it represents a UML SD. The notion of a construct unifying a set of constructs into a coherent whole which could be perceived as a diagram existed in UML 1 [UML1.4.2, p128], but the idea has been abandoned in UML 2. The unifying construct for UML 1 SD was Collaboration, which is the reason it is used by various UML tools (EA, Eclipse UML, see 2.7) and in the UML AST as the entry point for SDs. Collaboration exist in UML 2 but is no longer used as suggested in the UML AST.

Collaboration, shown in Listing E.15, consist of ownedBehavior: set of Interaction which in turn has the following:
lifelines : set of Lifeline: Represents an instance of a class. See section E.2.2.

fragments : set of InteractionFragment: An interaction fragment is a piece of an interaction. See section E.2.3.

messages : seq of Message: A Message defines a particular communication between Lifelines of an Interaction. See section E.2.4.

---

Collaboration ::
  ownedBehavior : set of Interaction;

Interaction ::
  name : String
  lifelines : set of LifeLine
  fragments : set of InteractionFragment
  messages : seq of Message;


---

E.2.2 LifeLine

Listing E.16 shows the type LifeLine, which represents an instance of a class via the optional represents : [Type] (see section E.1.3). If Lifeline do not reference a class it will be ignored during a transformation.

---

LifeLine ::
  name : String
  represents : [Type];

Listing E.16: LifeLine.

---

E.2.3 InteractionFragment

The type InteractionFragment shown in Listing E.2.3 defines the following:

OccurrenceSpecification: It is the basic semantic unit of Interactions. They give meaning to an Interaction by the sequence of occurrences (e.g. events) they specify.

InteractionOperand: It is contained in a CombinedFragment and represents one operand of the expression given by the enclosing CombinedFragment. See section E.2.3
CombinedFragment: It defines a boolean expression by an interaction operator and operand(s). See section E.2.3

ExecutionSpecification: It is a specification of the execution of a unit of behavior or action within a Lifeline.

Listing E.17 also shows the definition of Mos and Bes, which are explained in section E.2.3 and E.2.3, respectively.

```
InteractionFragment = OccurrenceSpecification | InteractionOperand | CombinedFragment | ExecutionSpecification;

OccurrenceSpecification = Mos;
-- Mos = MessageOccurrenceSpecification

ExecutionSpecification = Bes;
-- Bes = BehaviorExecutionSpecification

Listing E.17: InteractionFragment, Mos and Bes.
```

**InteractionOperand**

The **InteractionOperand of the InteractionFragment** consist of the following:

- **name**: String: The name of the fragment.
- **fragments**: seq of InteractionFragment: The operand represents a part of an interaction.
- **guard**: [InteractionConstraint]: It is an optional boolean expression that guards an operand in a CombinedFragment.
- **operand**: seq of InteractionOperand: A sequence of interaction operands see listing E.21. The operand it the fragment linked to a message.
- **covered**: set of LifeLine: A set of lifelines covered by the fragment.

```
InteractionOperand ::
  name : String
  fragments: seq of InteractionFragment
  covered : set of Mos
  guard : [InteractionConstraint];
```
### Listing E.18: InteractionOperand.

**InteractionConstraint ::**

minint : [ValueSpecification]

maxint : [ValueSpecification];

### Listing E.19: InteractionConstraint.

**CombinedFragment**

The CombinedFragment of the InteractionFragment consist of the following:

**interactionOperator : InteractionOperatorKind**

It is an enumeration designating the different kinds of operators of CombinedFragments. Only alt and loop are used in the UML AST.

**CombinedFragment ::**

name : String

interactionOperator : InteractionOperatorKind

operand : seq of InteractionOperand

covered : set of LifeLine;

### Listing E.20: CombinedFragment.

**InteractionOperatorKind = <ALT> | <LOOP>;**

### Listing E.21: InteractionOperatorKind.

**Mos and CallEvent**

The type Mos is an abbreviation of MessageOccurrenceSpecification. The reason it redefines OccurrenceSpecification in Listing E.17 is because it inherits OccurrenceSpecification in the UML specification.

Mos specifies the occurrence of event and in effect is the end of a Message, as shown in Listing E.22. The Mos has the optional event : CallEvent which is a specification of the reception of a request to invoke a specific operation. The Mos at the sending end of a Message has its event set to nil.
E.2. SEQUENCE DIAGRAM

---

```
Mos ::
  name : String
  message : [Message]
  covered : LifeLine
  event : [CallEvent];

CallEvent ::
  operation : Operation;
```

Listing E.22: Mos and CallEvent.

**Bes**

The type **Bes** is an abbreviation of **BehaviorExecutionSpecification**. The reason it redefines **ExecutionSpecification** in Listing E.17 is because it inherits **ExecutionSpecification** in the UML specification.

**Bes**, shown in Listing E.23, represents the execution of a behavior on the same **Lifeline**. The notation is the thin rectangular shapes attached to the dotted, vertical line stemming from the **Lifeline**. The **Bes** has the following:

- `startOs : OccurrenceSpecification`: It references the **Mos** which designates the start of the behavior.

- `finishOs : OccurrenceSpecification`: It references the **Mos** which designates the finish of the behavior.

```
Bes ::
  name : String
  startOs : OccurrenceSpecification
  finishOs : OccurrenceSpecification
  covered : set of LifeLine;
```

Listing E.23: Bes.

---

**E.2.4 Message**

Listing E.24 shows the type **Message** of the **Interaction**, which consist of the following:

- `messageKind : MessageKind`: It is an enumerated type that identifies the type of message. Only **complete** and **unknown** have been included. Even though
unknown does not occur, it is constructed as a union type to be up front with future development.

messageSort : MessageSort: It is an enumerated type that identifies the type of communication action that was used to generate the Message. Only syncCall and asyncCall have been included.

sendEvent : Mos: References the specification of the sending of the Message.

receiveEvent : Mos: References the specification of the reception of the Message.

<table>
<thead>
<tr>
<th>Message ::</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : String</td>
</tr>
<tr>
<td>messageKind : MessageKind</td>
</tr>
<tr>
<td>messageSort : MessageSort</td>
</tr>
<tr>
<td>sendEvent : Mos</td>
</tr>
<tr>
<td>sendReceive : Mos</td>
</tr>
<tr>
<td>argument : seq of ValueSpecification;</td>
</tr>
</tbody>
</table>

MessageKind = <COMPLETE> | <UNKNOWN>;
MessageSort = <SYNCHCALL> | <ASYNCHCALL>;

Listing E.24: Message, MessageKind and MessageSort.

E.3 UML Specification Citations

This section contains references to the UML meta-classes mentioned in section E, sorted alphabetically.

- Association [UMLSuperstructure2.1.2, p55]
- BehaviorExecutionSpecification (Bes) [UMLSuperstructure2.1.2, p483]
- CallEvent [UMLSuperstructure2.1.2, p450]
- Class [UMLSuperstructure2.1.2, p66]
- Collaboration [UML1.4.2, p128] and [UMLSuperstructure2.1.2, p184]
- CombinedFragment [UMLSuperstructure2.1.2, p483]
- Constraint [UMLSuperstructure2.1.2, p74]
- ExecutionSpecification [UMLSuperstructure2.1.2, p494]
- Interaction [UMLSuperstructure2.1.2, 497]
• InteractionConstraint [UMLSuperstructure2.1.2, p500]
• InteractionFragment [UMLSuperstructure2.1.2, p501]
• InteractionOperand [UMLSuperstructure2.1.2, p501]
• InteractionOperatorKind [UMLSuperstructure2.1.2, p502]
• Lifeline [UMLSuperstructure2.1.2, p506]
• LiteralInteger [UMLSuperstructure2.1.2, p107]
• LiteralString [UMLSuperstructure2.1.2, p108]
• Message [UMLSuperstructure2.1.2, p507]
• MessageKind [UMLSuperstructure2.1.2, p511]
• MessageSort [UMLSuperstructure2.1.2, p512]
• MultiplicityElement [UMLSuperstructure2.1.2, p110]
• OccurrenceSpecification [UMLSuperstructure2.1.2, p512]
• Operation [UMLSuperstructure2.1.2, p119]
• parameter [UMLSuperstructure2.1.2, p137]
• ParameterDirectionKind [UMLSuperstructure2.1.2, p138]
• Property [UMLSuperstructure2.1.2, p139]
• string [UMLSuperstructure2.1.2, p633]
• TemplateParameter [UMLSuperstructure2.1.2, p643]
• TemplateSignature [UMLSuperstructure2.1.2, p645]
• Type [UMLSuperstructure2.1.2, p151]
• ValueSpecification [UMLSuperstructure2.1.2, p154]
• VisibilityKind [UMLSuperstructure2.1.2, p155]
• MessageOccurrenceSpecification (Mos) [UMLSuperstructure2.1.2, p511]
Appendix F

Model coverage

In this chapter the model coverage is shown as a set of mathematical formatted VDM classes representing the core features of the transformation which includes Vdm2UML, Uml2Vdm and other core classes participating in the transformation. The coverage shown in this chapter is the result of a test suite which including a transformation test for each rule from chapter 4 and chapter 6. None executed paths are marked as gray and in the beginning of each section a comment list is shown. In the bottom of each section a table is provided showing the name, number of calls and coverage for each operation or function in the listed classes. The coverage of the model is only meant to illustrate that the transformation have been tested. From the coverage tables it can be seen that all involved class in a transformation has a coverage in the interval 90 to 99 percentage except the Oml2VppVisitor because it only implements a sub par of its base class. It is a fact that bugs exist in VDMTools which makes the coloring of the model e.g. isofclass colored wrong and there by calculates the coverage false. The complete model can be found on the attached CD-ROM.

F.1 Transforming from VDM to UML

In this section classes used to transform a VDM model into a UML model are shown.

F.1.1 Transformation from VDM to UML (Vdm2Uml)

The transformation from the OML AST to the UML AST.

init Convert a OmlSpecification to a UML Model.

build.uml Create the main UML model from a OML specification.

build.Class Convert a OML class to a UML class. This includes the creation of properties and associations.
**getGenericTypes**  Get template signature from OML types. Finds the template parameters.

**getSuperClasses**  Get super classes from a OML inheritance class. Returns nil if no super classes are found.

**hasSubclassResponsibilityDefinition**  Test if a OML Class is abstract in UML. Checks for a is sub class responsibility operation. True if found.

**build_def_b**  Proxy operation. Used to explicit set the type before redirecting operation call. This is introduced because of VDM Java gen limitation of type cast in Java.

**build_def_block (IOmlInstanceVariableDefinitions)**  Create properties from OML instance variables.

**buildVariable**  Create a UML property from a OML instance variable definition. If the instance variable is not mapped to a UML property nil is returned. This is the case if the property should be represented as an Association.

**getDefaultValue**  If a default value is defined for a instance variable it is returned as a string else nil.

**convertScopeToVisibility**  Convert OML visibility to UML visibility.

**build_def_block (IOmlValueDefinitions)**  Create UML properties from a value definition.

**buildValue**  Create a UML property from a Value definition. If the value is not presented as a UML property nil is returned. Would be the case if mapped as an Association.

**build_def_block (IOmlTypeDefinitions)**  Convert OML type definitions to a UML owned type.

**build_def_block (IOmlOperationDefinitions)**  Convert OML operations to UML owned operations.

**buildOperation**  Convert a OML operation definition to a UML definition. All parameters are ignored - Project time limitation.

**build_def_block (IOmlFunctionDefinitions)**  Convert OML functions to UML owned operations.

**buildFunction**  Convert a OML functions definition to a UML definition. All parameters are ignored - Project time limitation.

**isSimpleType**  Check if a OML type is mapped to a UML simple type.

**GetSimpleTypeName**  Get a simple type name.
CreateAssociationFromProperty  Create an Association from a UML property and a
OML type. The association is store in instance variable in the class.

CreateAssociationFromPropertyGeneral  Create a UML association from a UML prop-
erty where the OML type contains a OML type name.

CreateAssociationFromPropertyProductType  Create a UML association from a UML
property where the OML type is a product type.

CreateAssociationFromPropertyUnionType  Create a UML association from a UML
property where the OML type is a union type.

CreateEndProperty  Create Association ends from a OML type. Ends constructed
from Product and Union type and the anonymus end at the property owner end
of a association.

GetNextId  Get a new Id.

class Vdm2Uml
types
  public  String = char*  
instance variables
  associations : IUmlAssociation-set := {};
  constraints : IUmlConstraint-set := {};
  runningId : N := 0;

operations
  public
  init : IOmlSpecifications → IUmlModel
  init (specs) △
     (   let model = build-uml (specs) in
         (   model.setDefinitions
             (model.getDefinitions () ∪
              associations ∪
              constraints) ;
             return model
         )
     );

  public
  build-uml : IOmlSpecifications → UmlModel
  build-uml (specs) △
    let classes = specs.getClassList (),
        uml-classes = [build-Class (classes (i)) | i ∈ inds classes] in
  return new UmlModel ("Root", elems uml-classes);
public

build-Class : IOMlClass → IUmlClass
build-Class (c) △
  let name = c.getIdentifier (),
inh : [IOMlInheritanceClause] = if c.hasInheritanceClause ()
  then c.getInheritanceClause ()
  else nil ,

body = c.getClassBody (),
isStatic = false,
isActive =
  card { body (i) |
    i ∈ inds body ·
    isofclass (IOMlThreadDefinition, body (i)) } >
0,
dBlock = |let dbs : IOMlDefinitionBlock =
  body (i) in
  build-def-b (dbs, name) |
  i ∈ inds body |,
dBlockSet = { d | d ∈ elems dBlock · d ≠ nil },
isAbstract = hasSubclassResponsibilityDefinition (body),
supers = getSuperClasses (inh),
visibility =
  new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPUBLIC),
templateParameters = getGenericTypes (c.getGenericTypes ()) in
return new UmlClass (name,
  dBlockSet,
  isAbstract,
  supers,
  visibility,
  isStatic,
  isActive,
  templateParameters);

public
getGenericTypes: IOmlType* \rightarrow [IUmlTemplateSignature]
genericTypes (genericTypes) \triangleq
if len genericTypes > 0 then return new UmlTemplateSignature
( {let tn: IOmlTypeName = t in
   new UmlTemplateParameter (tn.getName ().getIdentifier ()) |
   t \in elems genericTypes})
else return nil;

public
getSuperClasses : [IOmlInheritanceClause] \rightarrow IUmlClassNameType*
getSuperClasses (inh) \triangleq
if inh = nil then return []
else let list = inh.getIdentifierList () in
   return [new UmlClassNameType (list (i)) | i \in inds list] :

public
hasSubclassResponsibilityDefinition : IOmlDefinitionBlock* \rightarrow B
hasSubclassResponsibilityDefinition (dBlock) \triangleq
let opList = conc [let op : IOmlOperationDefinitions = dBlock (i) in
   op.getOperationList () | i \in inds dBlock] · isofclass (IOmlOperationDefinitions, dBlock (i))],
   hasIsSubClassResp =
   {let explicitOp : IOmlExplicitOperation = opList (i).getShape () in
    explicitOp.getBody ().getSubclassResponsibility () |
    i \in inds opList} · isofclass (IOmlExplicitOperation, opList (i).getShape ())} in
return \exists e \in hasIsSubClassResp \cdot e = true;

private
build-def-b : IOmlDefinitionBlock \times String \rightarrow [IUmlDefinitionBlock]
build-def-b (block, owner) \triangleq
cases (true):
   (isofclass (IOmlInstanceVariableDefinitions, block)) \rightarrow
      let tmp : IOmlInstanceVariableDefinitions = block in
      build-def-block(tmp, owner),
   (isofclass (IOmlValueDefinitions, block)) \rightarrow
      let tmp : IOmlValueDefinitions = block in
      build-def-block(tmp, owner),
   (isofclass (IOmlTypeDefinitions, block)) \rightarrow
      let tmp : IOmlTypeDefinitions = block in
      build-def-block(tmp, owner).
(isofclass (IOmlOperationDefinitions, block)) →
  let tmp : IOmlOperationDefinitions = block in
  build-def-block(tmp, owner),
(isofclass (IOmlFunctionDefinitions, block)) →
  let tmp : IOmlFunctionDefinitions = block in
  build-def-block(tmp, owner),
  others → return nil
end;

public
  build-def-block : IOmlInstanceVariableDefinitions × String →
  IUmlOwnedProperties
build-def-block (v, owner) △
  let q = v.getVariablesList (),
  props = [buildVariable (q (i), owner) |
    i ∈ inds q ·
    isofclass (IOmlInstanceVariable, q (i))] in
  return new UmlOwnedProperties ({p | p ∈ elems props ·
    p ≠ nil });

public
  buildVariable : IOmlInstanceVariable × String → [IUmlProperty]
bUILDVariable (var, owner) △
  let access = var.getAccess (),
  scope = access.getScope (),
  assign = var.getAssignmentDefinition (),
  isStatic = access.getStaticAccess (),
  name = assign.getIdentifier (),
  visibility = convertScopeToVisibility (scope),
  omlType = assign.getType (),
  multiplicity = Vdm2UmlType'.extractMultiplicity (omlType),
  type = Vdm2UmlType'.convertPropertyType (omlType, owner),
  isReadOnly = false,
  default : [String] = if assign.hasExpression ()
    then getDefaultValue (assign.getExpression ())
    else nil ,
  isComposite = false,
  isDerived = false,
**F.1. TRANSFORMING FROM VDM TO UML**

```plaintext
qualifier : [IUmlType] = Vdm2UmlType'getQualifier (omlType) in
(dcl property : IUmlProperty := new UmlProperty (name, visibility, multiplicity, type, isReadOnly, default, isComposite, isDerived, isStatic, owner, qualifier);

if ¬ isSimpleType (omlType) then ( CreateAssociationFromProperty (property, omlType) ; return nil )
else return property );

public
getDefaultValue : IOmlExpression ⇄ [String]
getDefaultValue (expression) ≜
cases true:
(isofclass (IOmlSymbolicLiteralExpression, expression)) →
(let se : IOmlSymbolicLiteralExpression = expression in
cases true:
(isofclass (IOmlTextLiteral, se.getLiteral ()) →
(let tx : IOmlTextLiteral = se.getLiteral () in
return tx.getVal ()
),
(isofclass (IOmlNumericLiteral, se.getLiteral ()) →
(let tx : IOmlNumericLiteral = se.getLiteral () in
return StdLib 'ToString [N] (tx.getVal ())
),
others → return nil
end
),

others → return nil
end
```
convertScopeToVisibility : IOmlScope → IUmlVisibilityKind
convertScopeToVisibility (sc) ≜
  let val : N = sc.getValue () in
  cases val :
    (OmlScopeQuotes'IQPUBLIC) →
      new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPUBLIC),
    (OmlScopeQuotes'IQPRIVATE),
    (OmlScopeQuotes'IQDEFAULT) →
      new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPRIVATE),
    (OmlScopeQuotes'IQPROTECTED) →
      new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPROTECTED)
end

operations
pub
build-def-block : IOmlValueDefinitions × String → IUmlOwnedProperties
build-def-block (v, owner) ≜
  let q = v.getValueList (),
    props = [buildValue (q (i), owner) | i ∈ inds q],
    propsNoNil = {p | p ∈ elems props · p ≠ nil } in
  return new UmlOwnedProperties ((propsNoNil));

buildValue : IOmlValueDefinition × String → [IUmlProperty]
buildValue (var, owner) ≜
  let access = var.getAccess (),
    scope = access.getScope (),
    shape = var.getShape (),
    isStatic = access.getStaticAccess (),
    patternIdent : IOmlPatternIdentifier = shape.getPattern (),
    name = patternIdent.getIdentifier (),
    visibility = convertScopeToVisibility (scope),
    multiplicity = Vdm2UmlType'extractMultiplicity (shape.getType ()),
    type = Vdm2UmlType'convertType (shape.getType ()),
    isReadOnly = true,
    default = getDefaultValue (shape.getExpression ()),
    isComposite = false,
    isDerived = false,
    qualifier : [IUmlType] = Vdm2UmlType'getQualifier (shape.getType ()),

F.1. TRANSFORMING FROM VDM TO UML

\[ om\text{llType} = \text{shape}.\text{getType}() \text{ in} \]
\[
( \text{dcl \ property : IUmlProperty : = new UmlProperty (name, visibility, multiplicity, type, isReadOnly, default, isComposite, isDerived, isStatic, owner, qualifier);} \]

if \( \neg \text{isSimpleType(omllType)} \)
then ( \text{CreateAssociationFromProperty(property, omllType); return nil}
)
else return property
);

public
\[ \text{build-def-block : IOmlTypeDefinitions} \times \text{String} \rightarrow \text{UmlNestedClassifiers} \]
\[ \text{build-def-block (td,-)} \triangleq \]
\[
\text{let } q = \text{td}.\text{getTypeList}(),
\text{tps} = [\text{buildType (q (i).getShape())} | i \in \text{inds } q \cdot
\text{isofclass (IOmlSimpleType, q (i).getShape())}] \text{ in}
\text{return new UmlNestedClassifiers (elems tps);} \]

public
\[ \text{buildType : IOmlSimpleType} \rightarrow \text{UmlType} \]
\[ \text{buildType (var)} \triangleq \]
\[
\text{return Vdm2UmlType\text{'s convertType (var}.\text{getType}());} \]

public
\[ \text{build-def-block : IOmlOperationDefinitions} \times \text{String} \rightarrow \text{UmlOwnedOperations} \]
\[ \text{build-def-block (opDef , owner)} \triangleq \]
\[
\text{let } ops : \text{IOmlOperationDefinition}^* = \text{opDef}.\text{getOperationList}() \text{ in}
\text{return new UmlOwnedOperations ([buildOperation (ops (i), owner) |}
\[ i \in \text{inds } ops}])); \]

public
\[ \text{buildOperation : IOmlOperationDefinition} \times \text{String} \rightarrow \text{UmlOperation} \]
\[ \text{buildOperation (op,-)} \triangleq \]
\[
\text{let } access = \text{op}.\text{getAccess}(), \]
scope = access.getScope (),
shape: IOMlExplicitOperation = op.getShape (),
isStatic = access.getStaticAccess (),
name = shape.getIdentifier (),
visibility = convertScopeToVisibility (scope),
multiplicity = new UmlMultiplicityElement (false, false, 1, 1),
type = Vdm2UmlType' convertType (shape.getType ()) in
return new UmlOperation (name,
    visibility, 
multiplicity,
    false,
    type,
    isStatic,
    nil);

public

build-def-block : IOMlFunctionDefinitions x String o → IUm1OwnedOperations 
build-def-block (opDef, owner) Δ 
let ops : IOMlFunctionDefinition* = opDef.getFunctionList () in 
return new UmlOwnedOperations ({buildFunction (ops (i), owner) | 
i ∈ inds ops});

public

buildFunction : IOMlFunctionDefinition x String o → IUm1Operation 
buildFunction (op, -) Δ 
let access = op.getAccess (),
    scope = access.getScope (),
    shape: IOMlExplicitFunction = op.getShape (),
isStatic = access.getStaticAccess (),
name = shape.getIdentifier (),
visibility = convertScopeToVisibility (scope),
multiplicity = new UmlMultiplicityElement (false, false, 1, 1),
type = Vdm2UmlType' convertType (shape.getType ()) in
return new UmlOperation (name,
    visibility, 
multiplicity,
    true,
    type,
    isStatic,
    nil) ;
isSimpleType : IOmlType → β
isSimpleType (t) Δ
cases true :
  (isofclass (IOmlInjectiveMapType, t)),
  (isofclass (IOmlGeneralMapType, t)),
  (isofclass (IOmlTypeName, t)),
  (isofclass (IOmlProductType, t)),
  (isofclass (IOmlUnionType, t)) → false,
  (isofclass (IOmlSetType, t)) →
    let t1 : IOmlSetType = t in
    isSimpleType (t1.getType ()),
  (isofclass (IOmlSeq0Type, t)) →
    let t1 : IOmlSeq0Type = t in
    isSimpleType (t1.getType ()),
  (isofclass (IOmlSeq1Type, t)) →
    let t1 : IOmlSeq1Type = t in
    isSimpleType (t1.getType ()),
  (isofclass (IOmlOptionalType, t)) →
    let t1 : IOmlOptionalType = t in
    isSimpleType (t1.getType ()),
  others → true
end;

private

GetSimpleTypeName : IUmlType → String
GetSimpleTypeName (t) Δ
cases true :
  (isofclass (IUmlBoolType, t)) → ("bool"),
  (isofclass (IUmlIntegerType, t)) → ("int"),
  (isofclass (IUmlCharType, t)) → ("char"),
  others → ("String")
end

operations

CreateAssociationFromProperty : IUmlProperty × IOmlType → ()
CreateAssociationFromProperty (property, omlType) Δ
cases true:
  (isofclass (IOmlProductType, omlType)) →
    CreateAssociationFromPropertyProductType (property, omlType),
  (isofclass (IOmlUnionType, omlType)) →
    CreateAssociationFromPropertyUnionType (property, omlType),
others \rightarrow CreateAssociationFromPropertyGeneral(property, omlType)
end;

public
CreateAssociationFromPropertyGeneral : IUmlProperty \times IOmlType \rightarrow ()

CreateAssociationFromPropertyGeneral (property, -) \triangleleft
let ownerClassName = if isofclass (IUmlClassNameType, property.getType ())
then let pcn : IUmlClassNameType = property.getType () in
pcn.getName ()
else GetSimpleTypeName (property.getType ()),
propOtherEnd = {new UmlProperty ("",
    new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPRIVATE'),
    nil ,
    new UmlClassNameType (property.getOwnerClass ()),
    nil ,
    nil ,
    false ,
    nil ,
    ownerClassName ,
    nil )} in

associations := associations \cup
    {new UmlAssociation (propOtherEnd, {property}, nil , GetNextId ())};

public
CreateAssociationFromPropertyProductType : IUmlProperty \times IOmlType \rightarrow ()

CreateAssociationFromPropertyProductType (property, omlType) \triangleleft
let name : String = property.getName (),
    prop : UmlProperty = property,
    props : IUmlProperty-set = \bigcup \{CreateEndProperty (p, name) | p \in \{omlType}\}.
    isofclass (IOmlProductType, p) \} in

( prop.setName("" );
if card props > 1
    then associations := associations \cup
        {new UmlAssociation (props, {prop}, nil , GetNextId ())};

public
CreateAssociationFromPropertyUnionType : IUmlProperty × IOmlType → ()

CreateAssociationFromPropertyUnionType (property, omlType) △
  let name : String = property.getName (),
    prop : UmlProperty = property,
    props : IUmlProperty-set = ∪ { CreateEndProperty (p, name) | p ∈ { omlType } ·
      isofclass (IOmlUnionType, p) } in
  ( prop.setName("" );
    if card props > 1
      then ( dcl assoc : IUmlAssociation-set := { new UmlAssociation ({p}, {prop}, nil, GetNextId () ) | p ∈ props } ;
        associations := associations ∪ assoc ;
        constraints := constraints ∪
          { new UmlConstraint ( [ a.getId () | a ∈ assoc ], new UmlLiteralString (" xor") ) } )
  ) ;

public

CreateEndProperty : IOmlType × String → IUmlProperty-set
CreateEndProperty (t, name) △
  ( if (isofclass (IOmlProductType, t))
    then ( let typedType : IOmlProductType = t in
      return CreateEndProperty (typedType.getLhsType (), name) ∪

      CreateEndProperty (typedType.getRhsType (), name)
    )
  )

else if (isofclass (IOmlUnionType, t))
  then ( let typedType : IOmlUnionType = t in
    return CreateEndProperty (typedType.getLhsType (), name) ∪

    CreateEndProperty (typedType.getRhsType (), name) )
else return { new UmlProperty (name, 
    new UmlVisibilityKind (UmlVisibilityKindQuotes' IQPRIVATE), 
    Vdm2UmlType'extractMultiplicity (t), 
    Vdm2UmlType'convertType (t), 
    nil, 
    nil, 
    false, 
    nil, 
    nil, 
    "Implementation prosponed", 
    Vdm2UmlType'getQualifier (t)) }
);

private

GetNextId : () → String
GetNextId () △
    (runningId := runningId + 1;
    return Util'ToString[N] (runningId)
)
end Vdm2Uml

Test Suite : vdm.tc
Class : Vdm2Uml

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdm2Uml'init</td>
<td>31</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'GetNextId</td>
<td>26</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'buildType</td>
<td>36</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'build-uml</td>
<td>31</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'buildValue</td>
<td>3</td>
<td>90%</td>
</tr>
<tr>
<td>Vdm2Uml'build-Class</td>
<td>62</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'build-def-b</td>
<td>35</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'isSimpleType</td>
<td>71</td>
<td>86%</td>
</tr>
<tr>
<td>Vdm2Uml'buildFunction</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'buildVariable</td>
<td>53</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'buildOperation</td>
<td>4</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'getDefaultValue</td>
<td>18</td>
<td>95%</td>
</tr>
<tr>
<td>Vdm2Uml'getGenericTypes</td>
<td>62</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'getSuperClasses</td>
<td>62</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'CreateEndProperty</td>
<td>30</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'GetSimpleTypeName</td>
<td>2</td>
<td>62%</td>
</tr>
<tr>
<td>Vdm2Uml'convertScopeToVisibility</td>
<td>62</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'CreateAssociationFromProperty</td>
<td>20</td>
<td>✓</td>
</tr>
</tbody>
</table>
## F.1. TRANSFORMING FROM VDM TO UML

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdm2Uml'hasSubclassResponsibilityDefinition</td>
<td>62</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'CreateAssociationFromPropertyGeneral</td>
<td>14</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'CreateAssociationFromPropertyUnionType</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'CreateAssociationFromPropertyProductType</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'build-def-block</td>
<td>9</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'build-def-block</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'build-def-block</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'build-def-block</td>
<td>4</td>
<td>✓</td>
</tr>
<tr>
<td>Vdm2Uml'build-def-block</td>
<td>15</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Total Coverage</strong></td>
<td></td>
<td><strong>97%</strong></td>
</tr>
</tbody>
</table>
F.1.2 VDM to UML type converter (Vdm2UmlType)

Class providing type conversion.

extractMultiplicity Extract multiplicity from type.

getQualifier Get qualifier from type.

convertType Convert a OML type to UML.

convertPropertyType Convert property.

class Vdm2UmlType
types
    public String = char*
operations
    public static extractMultiplicity : IOmlType △ → IUmlMultiplicityElement
    extractMultiplicity (t) △
        (  dcl isOrdered : B := false,
           isUnique : B := true,
           lower : N := 1,
           upper : [N] := 1;
        cases true:
           (isofclass (IOmlSetType, t)) →
               (   upper := nil ;
                   lower := 0;
                   isOrdered := false
               ),
           (isofclass (IOmlSeq0Type, t)) →
               (   lower := 0;
                   upper := nil ;
                   isOrdered := true;
                   isUnique := false
               ),
           (isofclass (IOmlSeq1Type, t)) →
               (   lower := 1;
                   upper := nil ;
                   isOrdered := true;
                   isUnique := false
               ),
           (isofclass (IOmlNullType, t)) →
               (   upper := nil ;
                   lower := 0;
                   isOrdered := true;
                   isUnique := true
               ),
           (isofclass (IOmlInterfaceType, t)) →
               (   upper := nil ;
                   lower := 0;
                   isOrdered := true;
                   isUnique := true
               ),
           (isofclass (IOmlUnknownType, t)) →
               (   upper := nil ;
                   lower := 0;
                   isOrdered := true;
                   isUnique := true
               );
        );
    );
(isofclass (IOMlGeneralMapType, t)),
(isofclass (IOMlInjectiveMapType, t)) →
  (  isOrdered := true;
      upper := nil;
      lower := 0;
      isUnique := false
    ),
(isofclass (IOMlOptionalType, t)) →
  (  upper := 1;
      lower := 0
  )
end;
return new UmlMultiplicityElement (isOrdered, isUnique, lower, upper)
)

functions
public static
  getQualifier : IOMlType → [IUmlType]
getQualifier (t) △
cases true :
  (isofclass (IOMlInjectiveMapType, t)) → let t1 : IOMlInjectiveMapType = t in
    convertType (t1.getDomType ()),
  (isofclass (IOMlGeneralMapType, t)) → let t1 : IOMlGeneralMapType = t in
    convertType (t1.getDomType ()),
  others → nil
end;
public static
convertType : IOmlType → IUmlType

convertType (t) △

cases true :
  (isofclass (IOmlBoolType, t)) → new UmlBoolType (),
  (isofclass (IOmlNat1Type, t)) → new UmlIntegerType (),
  (isofclass (IOmlNatType, t)) → new UmlIntegerType (),
  (isofclass (IOmlIntType, t)) → new UmlIntegerType (),
  (isofclass (IOmlRealType, t)) → new UmlUnlimitedNatural (),
  (isofclass (IOmlCharType, t)) → new UmlCharType (),
  (isofclass (IOmlTokenType, t)) → new UmlIntegerType (),
  (isofclass (IOmlSetType, t)) → let t1 : IOmlSetType = t in
    convertType (t1.getType ()),
  (isofclass (IOmlSeq0Type, t)) → let t1 : IOmlSeq0Type = t in
    convertType (t1.getType ()),
  (isofclass (IOmlSeq1Type, t)) → let t1 : IOmlSeq1Type = t in
    convertType (t1.getType ()),
  (isofclass (IOmlInjectiveMapType, t)) → let t1 : IOmlInjectiveMapType = t in
    convertType (t1.getRngType ()),
  (isofclass (IOmlGeneralMapType, t)) → let t1 : IOmlGeneralMapType = t in
    convertType (t1.getRngType ()),
  (isofclass (IOmlEmptyType, t)) → nil ,
  (isofclass (IOmlOptionalType, t)) → let t1 : IOmlOptionalType = t in
    convertType (t1.getType ()),
  (isofclass (IOmlTypeName, t)) → let a : IOmlTypeName = t in
    new UmlClassNameType (a.getName ().getIdentifier ()),

  others → nil
end;

public static
convertPropertyType : IOmlType × String → IUmlType

convertPropertyType (t, owner) △

let ty = convertType (t) in
if ty = nil
then new UmlClassNameType (owner)
else ty
end Vdm2UmlType

Test Suite : vdm.tc
Class : Vdm2UmlType

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdm2UmlType’convertType</td>
<td>167</td>
<td>87%</td>
</tr>
<tr>
<td>Vdm2UmlType’getQualifier</td>
<td>74</td>
<td>✓</td>
</tr>
</tbody>
</table>
## F.1. TRANSFORMING FROM VDM TO UML

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdm2UmlType`convertPropertyType</td>
<td>53</td>
<td>√</td>
</tr>
<tr>
<td>Vdm2UmlType`extractMultiplicity</td>
<td>74</td>
<td>91%</td>
</tr>
<tr>
<td>Total Coverage</td>
<td></td>
<td>90%</td>
</tr>
</tbody>
</table>
F.1.3 Serilize the UML AST to XML with EA support (Uml2XmiEAxml)

Serialize the UML AST into a abstract XMI model.

**Save** Save UML model to file.

**CreateXmlFile** Create and construct XML model.

**AddDefinitions** Add definitions to XML model.

**AddConstraint** Add constraint to XML doc.


**GenerateClassIds** Create map of class names to class id in model.

**AddClass** Add class to XML model.

**AddProperties** Add properties to XML model.

**AddOperations** Add operations to XML model.

**AddAssociation** Add association to XML model.

**AddAssociationMp** Add multiplicity element to XML model.

**AddGeneralization** Add generalization to XML model.

**AddTemplates** Add template signature to XML model.

**GetUmlPrimitiveTypeId** Get id of UML primitive type used in XML model.

**AddStdTypes** Add primitive types to XML model.

**AddPrimitiveType** Add primitive type to XML model returns the id of the type added.

**GetTypeId** Get id in XML model of a UML type.

**GetNextId** Get next id for use in XML model.

**GetVisibilityKind** Get visibility name form visibility type. The name to use in the XML model.

class Uml2XmiEAxml is subclass of Uml2Xmi types
public
QualifierInfo :: AssociationId : String
  ClassId : String
  QualifierName : String;

public

QualifierEnd :: Property : IUmIProperty
  IsNavigable : □

values

  ID-TAG : String = "xmi:id";
  ownedMemberElementName : String = "ownedMember"

instance variables

protected id : Z := 1;
protected packageId : Z := 0;
protected classes : String → String := {→};
protected primitiveTypes : String ←→ String := {→→};
protected associationIdMap : String → String := {→→};
protected oe : String := "";
protected extensionTemplateClasses : String-set := {};
protected extensionConstrainElem : String-set := {};
protected extensionConectorNonNavigable : String-set := {};
protected extensionConectorQualifier : QualifierInfo-set := {};

operations

public

Save : char+ × IUmIModel → ()
Save (fileName, model) Δ
  ( dcl xmlVisitor : XmlFileOutputVisitor := new XmlFileOutputVisitor ();
    CreateXmlFile(model);
    Util.Clear();
    xmlVisitor.setEncoding("UTF-8");
    xmlVisitor.VisitXmlDocument(doc);
    Util.SaveBuf(fileName)
    );

protected
CreateXmlFile : IUmlModel → ()
CreateXmlFile (m) \(\triangleq\)

\[
\begin{align*}
oe & := \text{ownedMemberElementName}; \\
doc & .\text{StartE}("xmi: XMI"); \\
& \quad \text{doc}.\text{StartA}("xmi: version", "2.1"); \\
& \quad \text{doc}.\text{StartA}("xmlns:uml", "http://schema.omg.org/spec/UML/2.0"); \\
& \quad \text{doc}.\text{StartA}("xmlns:xmi", "http://schema.omg.org/spec/XMI/2.1"); \\
& \quad \text{doc}.\text{StartE}("xmi: Documentation"); \\
& \quad \text{doc}.\text{StartA}("xmi: Exporter", "Enterprise Architect"); \\
& \quad \text{doc}.\text{StartA}("xmi: ExporterVersion", "6.5"); \\
& \quad \text{doc}.\text{StopE}("xmi: Documentation"); \\
& \quad \text{doc}.\text{StartA}("name", m.get\text{Name}()); \\
& \quad \text{doc}.\text{StartA}(ID-TAG, GetNextId()); \\
& \quad \text{doc}.\text{StartE}(oe); \\
& \quad \text{doc}.\text{StartA}("xmi: type", "uml: Package"); \\
& \quad \text{doc}.\text{StartA}(ID-TAG, GetId(packageId)); \\
& \quad \text{doc}.\text{StartA}("name", "VDM Generated model"); \\
& \quad \text{AddStdTypes}(); \\
& \quad \text{GenerateClassIds}(m.getDefinitions()); \\
& \quad \text{AddDefinitions}(\{d \mid d \in m.getDefinitions() \cdot (\text{isofclass (), IUmlClass})d\}); \\
& \quad \text{AddDefinitions}(\{d \mid d \in m.getDefinitions() \cdot (\text{isofclass (), IUmlAssociation})d\}); \\
& \quad \text{AddDefinitions}(\{d \mid d \in m.getDefinitions() \cdot (\text{isofclass (), IUmlConstraint})d\}); \\
& \quad \text{doc}.\text{StopE}(oe); \\
& \quad \text{doc}.\text{StopE}("uml: Model"); \\
& \quad \text{AddExtention}(); \\
& \quad \text{doc}.\text{StopE}("xmi: XMI")
\end{align*}
\]

private

AddDefinitions : IUmlModelElement-set → ()
AddDefinitions (defs) \(\triangleq\)

\[
\begin{align*}
& \text{for all } c \in \{d \mid d \in \text{defs}\} \\
& \quad \text{do ( cases true:} \\
& \quad \quad (\text{isofclass (IUmlConstraint, c)} \rightarrow \text{AddConstraint}(c), \\
& \quad \quad (\text{isofclass (IUmlAssociation, c)} \rightarrow \text{AddAssociation}(c), \\
& \quad \quad (\text{isofclass (IUmlClass, c)} \rightarrow \text{AddClass}(c) \\
& \quad \quad \text{end}) \\
\end{align*}
\]

private
AddConstraint : IUmlConstraint → ()
AddConstraint (c) \triangleright
( doc.StartE(oe);
  doc.StartA("xmi : type", "uml : Constraint");
  let constrainId = GetNextId () in
    ( extensionConstrainElem := extensionConstrainElem ∪ {constrainId};
      doc.StartA(ID-TAG, constrainId)
    );
  for all a ∈ c.getConstraintElements ()
    do ( doc.StartE("constrainedElement");
          doc.StartA("xmi : idref", associationIdMap (a));
          doc.StopE("constrainedElement")
        );
  doc.StartE("specification");
  doc.StartA("xmi : type", "uml : OpaqueExpression");
  doc.StartA(ID-TAG, GetNextId ());
  doc.StopE("specification");
  doc.StartE("body");
  doc.StartD(if isofclass (IUmlLiteralString, c.getSpecification ())
    then let spec : IUmlLiteralString = c.getSpecification () in
      spec.getValue ()
    else ""
  );
  doc.StopE("body");
  doc.StopE("specification");
  doc.StopE(oe)
);

private
AddExstention : () → ()
AddExstention () \triangleright
( doc.StartE("xmi : Extension");
  doc.StartA("extender", "Enterprise Architect");
  doc.StartA("extenderID", "6.5");
  doc.StartE("elements");
  for all c ∈ extensionTemplateClasses
do ( doc.StartE("element") ;
    doc.StartA("xmi : idref", classes(c)) ;
doc.StartA("xmi : type", "uml : Class") ;
doc.StartA("name", c) ;
doc.StartA("scope", "public") ;
doc.StartE("properties") ;
doc.StartA("sType", "Class") ;
doc.StartA("nType", "1") ;
doc.StopE("properties") ;
doc.StopE("element") ) ;
for all constrainId ∈ extensionConstrainElem
    do ( doc.StartE("element") ;
        doc.StartA("xmi : idref", constrainId) ;
        doc.StartA("xmi : type", "uml : Constraint") ;
doc.StartA("scope", "public") ;
doc.StartE("properties") ;
doc.StartA("documentation", "xor") ;
doc.StartA("isSpecification", "false") ;
doc.StartA("sType", "Constraint") ;
doc.StartA("nType", "2") ;
doc.StartA("scope", "public") ;
doc.StopE("properties") ;
doc.StopE("element") ) ;
doc.StopE("elements") ;
doc.StartE("diagrams") ;
for all constrainId ∈ extensionConstrainElem
do (  
doc.StartE("diagram");
  doc.StartA(ID-TAG, GetNextId());
  doc.StartE("model");
  doc.StartA("package", GetId(packageId));
  doc.StartA("localID", "24");
  doc.StartA("owner", GetId(packageId));
  doc.StopE("model");
  doc.StartE("properties");
  doc.StartA("name", "Constrain diagram \{constrainId\}");
  doc.StartA("type", "Logical");
  doc.StopE("properties");
  doc.StartE("elements");
  doc.StartA("element");
  doc.StartA("geometry", "Left = 100; Top = 100; Right = 100; Bottom = 100; ");
  doc.StartA("subject", constrainId);
  doc.StartA("seqno", "1");
  doc.StartA("style", "DUID = AE8AC20D; ");
  doc.StopE("element");
  doc.StopE("elements");
  doc.StopE("diagram");
));
  doc.StopE("diagrams");
  doc.StartE("connectors");
  for all associationEndId ∈ extensionConectorNonNavigable
    do (  
      doc.StartE("connector");
      doc.StartA(xmi:idref, associationEndId);
      doc.StartE("properties");
      doc.StartA("ea_type", "Association");
      doc.StartA("direction", "Unspecified");
      doc.StopE("properties");
      doc.StopE("connector");
    )
  );
  for all qualifier ∈ extensionConectorQualifier
do (  
  doc.StartE("connector");
  doc.StartA("xmi:idref", qualifier.AssociationId);
  doc.StartE("target");
  doc.StartA("xmi:idref", qualifier.ClassId);
  doc.StartE("constraints");
  doc.StartA("qualifier", qualifier.QualifierName);
  doc.StopE("constraints");
  doc.StopE("target");
  doc.StopE("connector")
); 
  doc.StopE("connectors");
  doc.StopE("xmi:Extension")
); 

protected

GenerateClassIds : IUmlModelElement-set → ()
GenerateClassIds (defs) △
(  for all \( c \in \{ d \mid d \in \text{defs} \} \)
  do (  cases true:
    (isofclass (IUmlClass, c)) →
      ( let cl : IUmlClass = c in
        classes := classes\{cl.getName () → GetNextId ()\}
      )
  end
))
); 

protected
**F.1. TRANSFORMING FROM VDM TO UML**

AddClass : IUmlClass $\xrightarrow{a}$ ()

AddClass (cl) $\triangle$

(doc.StartE(oe);
  doc.StartA("isAbstract", Util.ToStringBool (cl.getIsAbstract ());
  doc.StartA("isAbstract", Util.ToStringBool (cl.getIsActive ());
  doc.StartA("isLeaf", "false");
  doc.StartA("name", cl.getName ());
  doc.StartA("visibility", "public");
  doc.StartA(ID-TAG, classes (cl.getName ());
  doc.StartA("xmi : type", "uml : Class");
AddPropeties(\{let d : IUmlOwnedProperties = df in
d.getPropetyList () | df $\in$ cl.getClassBody ()-iscofclass (IUmlOwnedProperties, df)});
AddOperstions(\{let d : IUmlOwnedOperations = df in
d.getOperationList () | df $\in$ cl.getClassBody ()-iscofclass (IUmlOwnedOperations, df)});
if len cl.getSuperClass () > 0
then AddGeneralization(cl.getSuperClass ());
if cl.hasTemplatesignature ()
then (AddTemplates(cl.getTemplatesignature ());
extensionTemplateClasses := extensionTemplateClasses $\cup$
  \{cl.getName ()
  )
  doc.StopE(oe)
);)

protected

AddPropeties : IUmlProperty-set $\xrightarrow{a}$ ()

AddPropeties (propeties) $\triangle$

(for all prop $\in$ propeties)
do (  
  doc.StartE("ownedAttribute") ;
  doc.StartA("name", prop.getName () ) ;
  doc.StartA("ownerScope", "instance");
  if prop.hasIsReadOnly ()
    then doc.StartA("isReadOnly", StdLib.ToStringBool (prop.getIsReadOnly ())) ;
  if prop.hasIsStatic ()
    then doc.StartA("isStatic", Util.ToStringBool (prop.getIsStatic ())) ;
  doc.StartA("visibility", GetVisibilityKind (prop.getVisibility ())) ;
  doc.StartA(ID-TAG, GetNextId ()) ;
  doc.StartA("xmi:type", "uml:Property");
  if prop.hasMultiplicity ()
    then (  
      doc.StartA("isOrdered", Util.ToStringBool (prop.getMultiplicity ().getIsOrdered ()) ;
      AddAssociationMp(prop.getMultiplicity ())
    ) ;
  if prop.hasDefault ()
    then (  
      doc.StartE("defaultValue") ;
      doc.StartA("xmi:type", "uml:LiteralString");
      doc.StartA(ID-TAG, GetNextId ()) ;
      doc.StartA("value", prop.getDefault () ) ;
      doc.StopE("defaultValue")
    ) ;
  doc.StartE("type") ;
  doc.StartA("xmi:idref", GetUmlPrimitiveTypeId (prop.getType ())) ;
  doc.StopE("type") ;
  doc.StopE("ownedAttribute")
);  

protected

AddOperations : IUmlOperation-set α ( )
AddOperations (ops) Δ  
  ( for all op ∈ ops
do (  
  doc.StartE("ownedOperation") ;  
  doc.StartA("isAbstract", "false") ;  
  doc.StartA("isLeaf", "false") ;  
  doc.StartA("isQuery", "false") ;  
  doc.StartA("name", op.getName()) ;  
  doc.StartA("ownerScope", "instance") ;  
  doc.StartA("visibility", GetVisibilityKind(op.getVisibility()) ;  
  doc.StartA(ID-TAG, GetNextId()) ;  
  doc.StartA("xmi : type", "uml : Operation") ;  
  doc.StopE("ownedOperation")  
)

protected
AddAssociation : IUmlAssociation $\to$ ()
AddAssociation (association) $\triangleq$
(  
  doc.StartE(oe) ;  
  doc.StartA("isAbstract", "false") ;  
  doc.StartA("isDerived", "false") ;  
  doc.StartA("isLeaf", "false") ;  
  doc.StartA("name", "") ;  
  let associationId = GetNextId() in  
  (  
    doc.StartA(ID-TAG, associationId) ;  
    associationIdMap := associationIdMap↑{association.getId() $\mapsto$ associationId} ;  
    doc.StartA("xmi : type", "uml : Association") ;  
    let unNamedProps : QualifierEnd* =
      Util.SetToSeq[QualifierEnd]  
      (  
        {mk-QualifierEnd (p, false) | p $\in$ association.getOwnedEnds()} .  
        len p.getName() = 0} $\cup$
        {mk-QualifierEnd (p, true) | p $\in$
          association.getOwnedNavigableEnds() .  
          len p.getName() = 0}  
      )
  ) ;  
) ;
namedProps =
Util\texttt{SetToSeq}[\texttt{QualifierEnd}]
\quad (\{\texttt{mk-QualifierEnd}(p, \texttt{false}) \mid p \in \text{association}.\text{getOwnedEnds}()\},
\quad \{\texttt{mk-QualifierEnd}(p, \texttt{true}) \mid p \in \text{association}.\text{getOwnedNavigableEnds}()\},
\quad \text{len}\ p.\text{getName}() > 0\}) \cup
\quad \{\texttt{mk-QualifierEnd}(p, \texttt{true}) \mid p \in \text{association}.\text{getOwnedNavigableEnds}()\},
\quad \text{len}\ p.\text{getName}() > 0\},
\quad \text{props} = (\text{unNamedProps} \setminus \text{namedProps}) \in
\quad \text{for all } i \in \text{inds props}
\quad \text{do let } prop = \text{props}(i) \text{ in}
\quad \quad (\text{if prop.\text{Property}.hasQualifier()}
\quad \quad \quad \text{then ( extensionConectorQualifier := extensionConectorQualifier} \cup
\quad \quad \quad \quad \{\texttt{mk-QualifierInfo}(\text{associationId},
\quad \quad \quad \quad \quad \quad \quad \text{\texttt{GetTypeId}(\text{new UmlClassNameType}(
\quad \quad \quad \quad \quad \quad \quad \text{prop.\text{Property}.getOwnerClass}()))},
\quad \quad \quad \quad \quad \quad \quad \text{\texttt{primitiveTypes}^{-1}}\}\)
\quad \quad \quad \text{);}
\quad \text{doc.StartE("ownedEnd");}
\quad \text{doc.StartA("aggregation","none");}
\quad \text{doc.StartA("association",associationId);}
\quad \text{doc.StartA("isNavigable",Util\texttt{ToStringBool}(prop.IsNavigable));}
\quad \text{if len (prop.\text{Property}.getName()) > 0}
\quad \text{then doc.StartA("name",prop.\text{Property}.getName());}
else doc.StartA("name", ");
    doc.StartA("visibility", GetVisibilityKind (prop.Property.getVisibility ()));
    let associationEndId = GetNextId () in
    (    doc.StartA(ID-TAG, associationEndId);
        doc.StartA("xmi : type", ", uml : Property");
        if prop.Property.hasMultiplicity ()
            then (    doc.StartA("isOrdered", Util.ToStringBool (prop.Property.getMultiplicity ()));
                AddAssociationMp(prop.Property.getMultiplicity ());
            ) ;
        if ¬prop.IsNavigable
            then extensionConectorNonNavigable := extensionConectorNonNavigable 
                {associationEndId};
            if prop.Property.hasIsStatic()
                then doc.StartA("isStatic", Util.ToStringBool (prop.Property.getIsStatic ()));
                doc.StartE("type");
                doc.StartA("xmi:idref", GetTypeId (prop.Property.getType ()));
                doc.StopE("type");
                doc.StopE("ownedEnd");
                doc.StartE("memberEnd");
                doc.StartA("xmi : idref", associationEndId);
                doc.StopE("memberEnd")
        ) ;
    doc.StopE(oe)
    );

protected AddAssociationMp : IUmlMultiplicityElement → ()
    AddAssociationMp (me) △
    (    doc.StartE("lowerValue");
        doc.StartA("value", Util.ToString[\N] (me.getLower ()));
        doc.StartA(ID-TAG, GetNextId ());
        doc.StopE("lowerValue");
        if me.hasUpper()
            then (    doc.StartE("upperValue");
                doc.StartA("value", Util.ToString[\N] (me.getUpper ()));
                doc.StartA(ID-TAG, GetNextId ());
                doc.StopE("upperValue")
            ) )
else (  
    doc.StartE(\"upperValue\") ;  
    doc.StartA("value", "\* ");  
    doc.StartA(ID-TAG, getNextId());  
    doc.StartA("xmi : type", "uml : LiteralString");  
    doc.StopE("upperValue")  
);  
protected  
AddGeneralization : IUmIClassNameType*  \( \rightarrow \) ()  
AddGeneralization (supers)  
(  
    for all \( a \in \) elems supers  
    do (  
        doc.StartE("generalization");  
        doc.StartA("xmi : type", "uml : Generalization");  
        doc.StartA(ID-TAG, getNextId());  
        doc.StartA("general", classes (\( a \).getName ()));  
        doc.StopE("generalization")  
    )  
);  
protected  
AddTemplates : IUmITemplateSignature  \( \rightarrow \) ()  
AddTemplates (tps)  
(  
    doc.StartE("ownedTemplateSignature");  
    doc.StartA("xmi : type", "uml : TemplateSignature");  
    doc.StartA(ID-TAG, getNextId());  
    for all \( a \in \) tps.getTemplateParameters ()  
    do (  
        dcl parameterId : String := getNextId ();  
    )  
);
### F.1. TRANSFORMING FROM VDM TO UML

```
protected

GetUmlPrimitiveTypeId : IUmlType → String
GetUmlPrimitiveTypeId (t) \(\triangleq\)
  cases true:
    (isofclass (IUml_BoolType, t)) → return primitiveTypes("bool"),
    (isofclass (IUml_IntegerType, t)) → return primitiveTypes("int"),
    (isofclass (IUml_UnlimitedNatural, t)) → return primitiveTypes("unlimitedNatural"),
    (isofclass (IUml_CharType, t)) → return primitiveTypes("char"),
    others → return primitiveTypes("String")
end;
```

protected
AddStdTypes : ()  \xrightarrow{o} ()
AddStdTypes () ⊳
  (  
    primitiveTypes := primitiveTypes †
    { "bool" \xrightarrow{} AddPrimitiveType ("bool")};
    primitiveTypes := primitiveTypes †
    { "int" \xrightarrow{} AddPrimitiveType ("int")};
    primitiveTypes := primitiveTypes †
    { "char" \xrightarrow{} AddPrimitiveType ("char")};
    primitiveTypes := primitiveTypes †
    { "unlimitedNatural" \xrightarrow{} AddPrimitiveType ("unlimitedNatural")};
    primitiveTypes := primitiveTypes †
    { "String" \xrightarrow{} AddPrimitiveType ("String")};
    primitiveTypes := primitiveTypes †
    { "NotSupportedType" \xrightarrow{} AddPrimitiveType ("NotSupportedType")}
  );

protected
AddPrimitiveType : String  \xrightarrow{o} String
AddPrimitiveType (typeName) ⊳
  (  
    doc.StartE (oe);
    doc.StartA("name", typeName);
    doc.StartA("visibility", "public");
    let tid = GetNextId () in
    (  
      doc.StartA(ID-TAG, tid);
      doc.StartA("xmi:type", "uml:Class");
      doc.StopE (oe);
      return tid
    )
  );

protected
GetTypeId : IUmlType  \xrightarrow{o} String
GetTypeId (t) ⊳
  cases true:
  (isofclass (IUmlClassNameType, t)) →
  let qc : IUmlClassNameType = t in
  (  
    if ∃ x ∈ dom classes · x = qc.getName ()
    then ( return classes (qc.getName ())
    )
  else ( return primitiveTypes ("NotSupportedType")
  );
(isofclass (IUmlBoolType, t)),
(isofclass (IUmlIntegerType, t)),
(isofclass (IUmlCharType, t)),
(isofclass (IUmlUnlimitedNatural, t)) → return GetUmlPrimitiveTypeId (t),
others → return primitiveTypes ("NotSupportedType")
end;

protected
GetNextId : () → String
GetNextId () △
  ( id := id + 1;
    return GetId (id)
  )

functions
protected
GetId : Z → String
GetId (idNum) △
  "VDM. " ∼ Util.ToString[Z] (idNum);

protected
GetVisibilityKind : IUmlVisibilityKind → String
GetVisibilityKind (visibility) △
cases visibility.getValue () :
  (UmlVisibilityKindQuotes'IQPUBLIC) → ("public"),
  (UmlVisibilityKindQuotes'IQPRIVATE) → ("private"),
  (UmlVisibilityKindQuotes'IQPROTECTED) → ("protected")
end

end Uml2XmiEAXml

Test Suite : vdm.tc
Class : Uml2XmiEAXml

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uml2XmiEAXml'Save</td>
<td>19</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'GetId</td>
<td>406</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'AddClass</td>
<td>38</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'GetNextId</td>
<td>383</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'GetTypeId</td>
<td>43</td>
<td>86%</td>
</tr>
<tr>
<td>Uml2XmiEAXml'AddStdTypes</td>
<td>19</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'AddProperties</td>
<td>38</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'AddTemplates</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'AddConstraint</td>
<td>2</td>
<td>98%</td>
</tr>
<tr>
<td>Uml2XmiEAXml'AddExtention</td>
<td>19</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml'AddOperations</td>
<td>38</td>
<td>✓</td>
</tr>
</tbody>
</table>
### APPENDIX F. MODEL COVERAGE

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uml2XmiEAXml`CreateXmlFile</td>
<td>19</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`AddAssociation</td>
<td>17</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`AddDefinitions</td>
<td>57</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`AddAssociationMp</td>
<td>53</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`AddPrimitiveType</td>
<td>114</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`GenerateClassIds</td>
<td>19</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`AddGeneralization</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`GetVisibilityKind</td>
<td>65</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2XmiEAXml`GetUmlPrimitiveTypeId</td>
<td>32</td>
<td>73%</td>
</tr>
<tr>
<td><strong>Total Coverage</strong></td>
<td></td>
<td><strong>98%</strong></td>
</tr>
</tbody>
</table>
F.2 Transforming UML to VDM

In this section classes used to transform a UML model into VDM is shown.

F.2.1 Convert XMI to a UML model (Xml2UmlModel)

Convert the XMI document to the UML AST.

VisitXmlDocument Create UML model form document. If a valid document is passed the result variable will contain the result parsed document.

extractClass Extract UML class from XML element.

build_defBlock Create property definitions form XML property element. Attribute type is property.

build_Property Create property from property element.

getDefaultValue Get default value. Initial value.

getMultiplicity Get multiplicity.

build_Operation Create operation from xml element element.

build_Constraint Create constraint from constraint elements.

build_Association Create association from association element.

lookUpType Get UmlType from XML type name.

VisitXmlEntity Visit element and build classes or association depended on the element type attribute.

hasAttribute Test if a element has an attribute with a specific name.

hasAttributeValue Test if a element has a specific element with a specific name and type.

isAttributeType Test if a element has a specific type. By looking up the type attribute.

getElementType Get the UML type attribute value or the name of the element if no such attribute exists.

g.getAttribute Get XML attribute from name. If unknown return nil.

GetVisibility Convert visibility name to UmlVisibility type.

build_Collaboration Create a Collaboration from a XML node.

build_Interaction Create a Collaboration from a XML node.
**build Message** Create a Collaboration from a XML node.

**build Fragment** Delegating the creation of a fragment to the responsible operation only needed because limitation in Java Code Gen.

**build Mos** Create MOS from XmlElement.

**build Bes** Create BES from XmlElement.

**build Combi** Create CombinedFragment from XmlElement.

**GetIntOperationKind** Get interaction kind from a string.

**build Operand** Create Operand from XmlElement.

**GetCoveredExtension** Get covered extension. Extracts covered string.

**GetCovered** Get covered seq of ids.

**GetGuard** Get guard from entity.

**getGuardConstraintValue** Get value of constraint on a guard.

**buildCallEventMap** Build id to callEvent map.

class Xml2UmlModel is subclass of XmlVisitor

types
  public String = char*

instance variables
  public result : [IUmlModel] := nil;
  primiticeTypes : String-set := {"char", "int", "bool", "String", "unlimitedNatural", "NotSupportedType"};
  classes : IUmlClass-set := {};
  associations : IUmlAssociation-set := {};
  constraints : IUmlConstraint-set := {};
  classesTypeMap : String m→ String := {ℰ→};
  collaborations : IUmlCollaboration-set := {};
  idToclassesMap : String m→ IUmlClass := {ℰ→};
  idToOperationMap : String m→ IUmlOperation := {ℰ→};
  idToCallEventMap : String m→ IUmlCallEvent := {ℰ→};
  lifeLineMap : String m→ IUmlLifeLine := {ℰ→};
  mosMap : String m→ IUmlMos := {ℰ→};
  besMap : String m→ IUmlBes := {ℰ→};
  combiMap : String m→ IUmlCombinedFragment := {ℰ→};

operations
public
Visit XmlDocument : XmlDocument \xrightarrow{a} ()

Visit XmlDocument (doc) \triangleq
t (dcl root : XmlEntity := hd doc.entities.entities, 
    firstPackageAndRoot : XmlEntity* := root.entities.entities \rightarrow [root],
    model : XmlEntity := hd [firstPackageAndRoot (i) | i \in inds firstPackageAndRoot .
        firstPackageAndRoot (i).name = "uml: Model"],
    package : XmlEntity := hd [model.entities.entities (i) | i \in inds model.entities.entities .
        isAttributeType (model.entities.entities (i), "uml: Package")]);
let pes = package.entities.entities in
  (classesTypeMap := classesTypeMap | {getAttribute (pes (i), "xmi:id").val → getAttribute (pes (i), "name").val | i ∈ inds pes ·getAttributeType (pes (i), "uml:Class")}) |
  (getAttribute (pes (i), "name").val ∉ primiticeTypes)};
  classes := classes∪{exstractClass (pes (i)) | i ∈ inds pes ·getAttributeType (pes (i), "uml:Class")∧
  (getAttribute (pes (i), "name").val ∉ primiticeTypes)};
  associations := associations∪{build-Association (pes (i)) | i ∈ inds pes ·getAttributeType (pes (i), "uml:Association")};
  constraints := constraints ∪ {build-Constraint (pes (i)) | i ∈ inds pes ·getAttributeType (pes (i), "uml:Constraint")};
  idToCallEventMap := buildCallEventMap ({} (pes (i)) | i ∈ inds pes ·getAttributeType (pes (i), "uml:CallEvent");
  collaborations := collaborations∪{build-Composition (pes (i)) | i ∈ inds pes ·getAttributeType (pes (i), "uml:Collaboration")};
  result := new UmlModel (getAttribute (package, "name").val,
  classes∪associations∪constraints∪
collaborations)
);
private
exstractClass : XmlEntity → IUmlClass
exstractClass (e) △
  let name : char* = getAttribute (e, "name").val,
  dBlocks : (IUmlDefinitionBlock-set) = build-defBlock (e, name),
  abstract : B = if hasAttribute (e, "isAbstract")
  then StdLib'StringToBool (getAttribute (e, "isAbstract").val)
  else false,
  supers : IUmlClassNameType* = [],
  visibility : (IUmlVisibilityKind) =
    new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPUBLIC),
isStatic : (B) =
false,
active : (B) = if hasAttribute (e, "isActive")
    then let a = getAttribute (e, "isActive") in
        if a ≠ nil
            then StdLib.StringToBool (a.val)
            else false
    else false
else false,
template : [IUmlTemplateSignature] = nil,
id : String = getAttribute (e, "xmi:id").val,
cls : IUmlClass =
    new UmlClass (name,
dBlocks,
abstract,
supers,
visibility,
isStatic,
active,
template) in
    ( idToclassesMap := idToclassesMap † {id ↦→ cls};
    return cls )
pre isAttributeType (e, "uml:Class");
private
build-defBlock : XmlEntity × String ⊸ (IUmlDefinitionBlock-set)
build-defBlock (e, name) △
    let eList = e.entities.entities,
        props = {build-Property (eList (i), name) |
            i ∈ inds eList · isAttributeType (eList (i), "uml:
        Property")},
        ops = {build-Operation (p) |
            p ∈ elems eList-isAttributeType (p, "uml:Operation")} in
    return {new UmlOwnedProperties (props)} ∪ {new UmlOwnedOperations (ops)};
private
build-Property : XmlEntity × String ⊸ IUmlProperty
build-Property (e, ownerClass) △
    let name : char* = getAttribute (e, "name").val,
        visibility : (IUmlVisibilityKind) = if hasAttribute (e, "visibility")
            then GetVisibility (getAttribute (e, "visibility"))
            else new UmlVisibilityKind (UmlVisibilityKindQuotes
        multiplicity : [IUmlMultiplicityElement] = getMultiplicity (e),
\begin{verbatim}
type : (IUmlType) = lookUpType (e),
isReadOnly : [B] =
    if hasAttribute (e, "isReadOnly")
    then StdLib 'StringToBool (getAttribute (e, "isReadOnly").val)
    else false,
default : [char*] = getDefaultValue (e),
isComposite : [B] = false,
isDerived : [B] = false,
isStatic : [B] = false,
qualifier : [IUmlType] =
    nil in
return new UmlProperty (name, visibility, multiplicity, type, isReadOnly, default, isComposite, isDerived, isStatic, ownerClass, qualifier)

pre  isAttributeType (e, "uml : Property") ;

private
getDefaultValue : XmlEntity \rightarrow [String]
getDefaultValue (e) \triangleq
let eList = e.entities.entities,
tmp = \{ getAttribute (el, "value").val | el \in elems eList.el.name = "defaultValue" \} in
if card tmp > 0
    then let t \in tmp in
        return t
    else return nil ;

private
getMultiplicity : XmlEntity \rightarrow [IUmlMultiplicityElement]
getMultiplicity (e) \triangleq
let eList = e.entities.entities,
lowerSet = \{ getAttribute (el, "value").val | el \in elems eList . el.name = "lowerValue" \land hasAttribute (e, "value") \},
upperSet = {getAttribute (el, "value").val | el ∈ elems eList ·
   el.name = "upperValue" ∧ hasAttribute (el, "value")} in
   if card upperSet = 0 ∧ card lowerSet = 0
   then return nil
   else let lower : N = if card lowerSet = 0
               then 0
               else let p ∈ lowerSet in
                      StdLib’StringToInt (p),
               upper : [N] = if card upperSet = 0
                      then nil
                      else let p ∈ upperSet in
                            StdLib’StringToInt (p) in
              return new UmlMultiplicityElement (false, false, lower, upper);

private
build-Operation : XmlEntity → IUmlOperation
build-Operation (e) △
   let name : char∗ = getAttribute (e, "name").val,
   visibility : (IUmlVisibilityKind) = GetVisibility (getAttribute (e, "visibility")),
   multiplicity : IUmlMultiplicityElement = new UmlMultiplicityElement (false, false, 0, 0),
   isQuery : B =
   if hasAttribute (e, "isQuery")
   then StdLib’StringToBool (getAttribute (e, "isQuery").val)
   else false,
   type : [IUmlType] = nil,
   isStatic : B = if hasAttribute (e, "isStatic")
   then StdLib’StringToBool (getAttribute (e, "isStatic").val)
   else false,
   id : String = getAttribute (e, "xmi:id").val,
   operation : IUmlOperation = new UmlOperation (name,
       visibility,
       multiplicity,
       isQuery,
       type,
       isStatic,
       nil ) in
   ( idToOperationMap := idToOperationMap†{id → operation};
     return operation
   )
pre isAttributeType (e, "uml:Operation")
functions
private
build-Constraint : XmlEntity → IUmlConstraint
build-Constraint (e) △
  letelist = e.entities.entities,
  ids = {getAttribute (p, "xmi:idref").val | p ∈ elems elist . hasAttribute (p, "xmi:idref")},
  specification : String = hd Util:SetToSeq[char*] ({p.data.data | p ∈ elems elist . p.name = "body")} in
  new UmlConstraint (ids, new UmlLiteralString (specification))
pre isAttributeType (e, "uml:Constraint")

operations
private

build-Association : XmlEntity → IUmlAssociation
build-Association (e) △
  letelist = e.entities.entities,
  props = {elist (i) | i ∈ indselist . isAttributeType (elist (i), "uml:Property")},
  ownedNavivableEnds : (IUmlProperty-set) = {build-Property (p, "") | p ∈ props . hasAttributeValue (p, "name", "")},
  one = Util:SetToSeq[IUmlProperty] (ownedNavivableEnds),
  ownerClassType = hd [one (i).getType () | i ∈ inds one],
  ownerClass = let ct : IUmlClassNameType = ownerClassType in
  ct.getName ().,
  ownedEnds : (IUmlProperty-set) = {build-Property (p, ownerClass) | p ∈ props . len getAttribute (p, "name").val > 0},
  name : char* = getAttribute (e, "name").val,
  id : String = getAttribute (e, "xmi:id").val in
  return new UmlAssociation (ownedEnds, ownedNavivableEnds, name, id)
pre isAttributeType (e, "uml:Association");

private

lookUpType : XmlEntity → IUmlType
lookUpType (e) △
  letelist = e.entities.entities,
  typeOption1 = if hasAttribute (e, "type")
  then {getAttribute (e, "type").val}
  else {},
typeOption2 = \{ \text{getAttribute} (\text{elist} (i), "xmi:idref").val \mid i \in \text{inds} \text{elist} \cdot \text{elist} (i).\text{name} = "type" \} \cup \\
\{ \text{a.val} \mid \text{a} \in \{ \text{getAttribute} (e, "type") \} \cdot \text{a} \neq \text{nil} \} \}

\text{in}
\text{let id} \in \text{typeOption1} \cup \text{typeOption2} \text{ in}
\text{let typeName = if id} \in \text{dom \ classesTypeMap}
\text{ then \ classesTypeMap (id) }
\text{ else \ nil \ in}
\text{cases \ typeName:}
\quad \text{nil} \rightarrow \text{return new UmlIntegerType ()},
\quad "\text{String}" \rightarrow \text{return new UmlStringType ()},
\quad "\text{int}" \rightarrow \text{return new UmlIntegerType ()},
\quad "\text{bool}" \rightarrow \text{return new UmlBoolType ()},
\quad "\text{char}" \rightarrow \text{return new UmlCharType ()},
\quad "\text{unlimitedNatural}" \rightarrow \text{return new UmlUnlimitedNatural ()},
\quad \text{others} \rightarrow \text{return new UmlClassNameType (typeName)}
\text{end};

\text{public}
\text{VisitXmlEntity : XmlEntity} \xrightarrow{\circ} ()
\text{VisitXmlEntity} (e) \triangleq
\begin{cases}
\quad \text{classes := classes} \cup \{ \text{extractClass (entity)} \mid \text{entity} \in \{ e \} \cdot}
\quad \quad \text{isAttributeType (entity, "uml:Class") \land}
\quad \quad \{ \text{getAttribute (entity, "name").val } \notin \text{primiticeTypes} \} ;
\quad \text{associations := associations} \cup \{ \text{build-Association (entity)} \mid \text{entity} \in}
\quad \quad \text{isAttributeType (entity, "uml:Association")} \}
\end{cases}

\text{functions}
\text{private}
\text{hasAttribute : XmlEntity} \times \text{String} \rightarrow \mathbb{B}
\text{hasAttribute} (e, \text{name}) \triangleq
\quad \text{let list} = e.\text{attributes.attributes} \text{ in}
\quad \exists i \in \text{inds list} \cdot \text{list} (i).\text{name} = \text{name};

\text{private}
\text{hasAttributeValue : XmlEntity} \times \text{String} \times \text{String} \rightarrow \mathbb{B}
\text{hasAttributeValue} (e, \text{name}, \text{val}) \triangleq
\quad \text{let list} = e.\text{attributes.attributes} \text{ in}
\quad \exists i \in \text{inds list} \cdot \text{list} (i).\text{name} = \text{name} \land \text{list} (i).\text{val} = \text{val};
\begin{align*}
isAttributeType : \text{XmlEntity} \times \text{String} \rightarrow \mathbb{B} \\
isAttributeType (e, \text{val}) \triangleq \\
\hspace{1cm} \text{hasAttributeValue} (e, \text{"xmi : type"}, \text{val});
\end{align*}

```java
private

gisElementType : \text{XmlEntity} \rightarrow \text{String}
gisElementType (e) \triangleq 
\hspace{1cm} \text{if getAttribute} (e, \text{"xmi : type"}) \neq \text{nil} \\
\hspace{2cm} \text{then getAttribute} (e, \text{"xmi : type"}).\text{val} \\
\hspace{2cm} \text{else e.name};
```

```java
private

ggetAttribute : \text{XmlEntity} \times \text{String} \rightarrow [\text{XmlAttribute}]
ggetAttribute (e, \text{name}) \triangleq 
\hspace{1cm} \text{let list = e.attributes.attributes,} \\
\hspace{2cm} \text{attList = [list (i) | i \in \text{inds list} \cdot \text{list (i).name = name}] in} \\
\hspace{2cm} \text{if len attList > 0} \\
\hspace{3cm} \text{then hd attList} \\
\hspace{3cm} \text{else nil ;}
```

```java
private static

GetVisibility : [XmlAttribute] \rightarrow \text{IUmlVisibilityKind}
GetVisibility (v) \triangleq 
\hspace{1cm} \text{if v \neq \text{nil}} \\
\hspace{2cm} \text{then cases v.val :} \\
\hspace{3cm} \text{"private"} \rightarrow \\
\hspace{4cm} \text{new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPRIVATE),} \\
\hspace{3cm} \text{"public"} \rightarrow \\
\hspace{4cm} \text{new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPUBLIC),} \\
\hspace{3cm} \text{"protected"} \rightarrow \\
\hspace{4cm} \text{new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPROTECTED)} \\
\hspace{2cm} \text{end} \\
\hspace{2cm} \text{else new UmlVisibilityKind (UmlVisibilityKindQuotes'IQPRIVATE)}
```

```java
operations

build-Collaboration : \text{XmlEntity} \rightarrow \text{IUmlCollaboration}
build-Collaboration (e) \triangleq 
\hspace{1cm} \text{let elist = e.entities.entities,} \\
\hspace{2cm} \text{interactions = \{build-Interaction (p) | p \in \text{elems elist-isAttributeType} (p, \text{"uml: Interaction"})\} in} \\
\hspace{2cm} \text{return new UmlCollaboration (interactions)} \\
\hspace{1cm} \text{pre isAttributeType} (e, \text{"uml : Collaboration"});
```

private
build-Interaction : XmlEntity ➝ IUmlInteraction
build-Interaction (e) △
  let elist = e.entities, props : String ➝ IUmlProperty = {getAttribute (p, "xmi:id").val ➝ build-Property (p, "") | p ∈ elems elist-isAttributeType (p, "uml:Property")} in
    (lifeLineMap := lifeLineMap†{getAttribute (p, "xmi:id").val ➝

    new UmlLifeLine (getAttribute (p, "name")).val,
    props (getAttribute (p, "represents").val).getType ()

    p ∈ elems elist ·
    p.name = "lifeline";

    let name : char* = getAttribute (e, "name").val,
    lifeLines : IUmlLifeLine-set = rng lifeLineMap,
    mosfragments : IUmlInteractionFragment-set =
      {build-Fragment (f) | f ∈ elems elist ·
       isAttributeType (f, "uml:MessageOccurrenceSpecification")},
    besfragments : IUmlInteractionFragment-set =
      {build-Fragment (f) | f ∈ elems elist ·
       isAttributeType (f, "uml:BehaviorExecutionSpecification")},
    combifragments : IUmlInteractionFragment-set =
      {build-Fragment (f) | f ∈ elems elist ·
       isAttributeType (f, "uml:CombinedFragment")},
    messages : IUmlMessage* =
      [build-Message (elist (i)) | i ∈ inds elist ·
       elist (i).name = "message"] in
    return new UmlInteraction (name,
    lifeLines,
    mosfragments∪besfragments∪combifragments,
    messages)
  )

pre isAttributeType (e, "uml:Interaction") ;

private

build-Message : XmlEntity ➝ IUmlMessage
build-Message (e) △
  let messageKind : IUmlMessageKind =
    new UmlMessageKind (UmlMessageKindQuotes'IQCOMPLETE),
  messageSort : IUmlMessageSort = new UmlMessageSort (UmlMessageSortQuotes'IQSY),
  mosSend : IUmlMos = mosMap (getAttribute (e, "sendEvent").val),
  mosRecive : IUmlMos = mosMap (getAttribute (e, "receiveEvent").val),
  args : (IUmlValueSpecification*) = [],
APPENDIX F. MODEL COVERAGE

```java
name : String = mosRecive.getEvent().getOperation().getName() in
return new UmlMessage (name,
  messageKind,
  messageSort,
  mosSend,
  mosRecive,
  args)
pre e.name = "message";

private
build-Fragment : XmlEntity ➝ IUmlInteractionFragment
build-Fragment (e) △
cases getElementType (e):
  "uml : MessageOccurrenceSpecification" ➝
  return build-Mos (e),
  "uml : BehaviorExecutionSpecification" ➝
  return build-Bes (e),
  "uml : CombinedFragment" ➝
  return build-Combi (e)
end
pre isAttributeType (e, "uml : MessageOccurrenceSpecification") ∨
  isAttributeType (e, "uml : BehaviorExecutionSpecification") ∨
  isAttributeType (e, "uml : CombinedFragment") ;

private
build-Mos : XmlEntity ➝ IUmlMos
build-Mos (e) △
let name = getAttribute (e, "name").val,
  message : IUmlMessage = nil,
  lifeLines : IUmlLifeLine = let l ∈ {lifeLineMap (c) | c ∈ elems GetCovered (e)} in l,
  id = getAttribute (e, "xmi : id").val,
  event : IUmlCallEvent = idToClallEventMap (getAttribute (e, "event").val),
  mos = new UmlMos (name,
    message,
    lifeLines,
    event) in
(  mosMap := mosMap ⧵ {id ↦ mos};
  return mos)
pre isAttributeType (e, "uml : MessageOccurrenceSpecification") ;

private
```

This section appears to discuss the implementation of a method related to UML (Unified Modeling Language) messages. It includes Java code for handling message occurrences and interactions, specifically focusing on how to create `UmlMessage` objects from elements of an XML entity. The code snippet demonstrates how to build messages and life lines from UML elements, with a focus on parameterizing the message kind and sorting. The method `build-Mos` is particularly highlighted, which constructs a `UmlMos` object based on the attributes and elements of an XML entity to represent UML messages.
F.2. TRANSFORMING UML TO VDM

build-Bes : XmlEntity $\rightarrow$ IUmlBes
build-Bes (e) $\triangle$

let name =getAttribute (e, "name").val,
    startOc : IUmlMos = mosMap (getAttribute (e, "start").val),
    finishOc : IUmlMos = mosMap (getAttribute (e, "finish").val),
    covered : IUmlLifeLine-set = \{lifeLineMap (c) \mid c \in \text{elems GetCovered} (e)\},
    id = getAttribute (e, "xmi : id").val,
    bes = new UmlBes (name,
        startOc,
        finishOc,
        covered) in
    (besMap := besMap $\triangle\{id \mapsto \{\}$; return bes)

pre isAttributeType (e, "uml : BehaviorExecutionSpecification") ;

private

build-Combi : XmlEntity $\rightarrow$ IUmlCombinedFragment
build-Combi (e) $\triangle$

let elist = e.entities.entities,
    name = getAttribute (e, "name").val,
    interactionOperatorKind : IUmlInteractionOperatorKind = getIntOperationKind (getAttribute (e, "interactionOperator").val),
    operands : IUmlInteractionOperand* = [build-Operand (elist (i)) | i \in \text{inds elist} \cdot elist (i).name = "operand"],
    covered : IUmlLifeLine-set = \{lifeLineMap (c) \mid c \in \text{elems GetCovered} (e)\},
    id = getAttribute (e, "xmi : id").val,
    combi = new UmlCombinedFragment (name,
        interactionOperatorKind,
        operands,
        covered) in
    (combiMap := combiMap $\triangle\{id \mapsto \{\}$; return combi)

pre isAttributeType (e, "uml : CombinedFragment")

functions

getIntOperationKind : String $\rightarrow$ IUmlInteractionOperatorKind
getIntOperationKind (text) $\triangle$

cases text :
"alt" $\rightarrow$ new UmlInteractionOperatorKind (UmlInteractionOperatorKindQuotes 'IQALT'
"loop" $\rightarrow$ new UmlInteractionOperatorKind (UmlInteractionOperatorKindQuotes 'IQLOOP'
end
APPENDIX F. MODEL COVERAGE

pre text = "alt" ∨ text = "loop"

operations
private

build-Operand : XmlEntity → IUmlInteractionOperand
build-Operand (e) ∆
  let eList = e.entities.entities,
  name : char* = "",
  fragments : IUmlInteractionFragment* = [],
  covered : IUmlMos-set = {mosMap (c) | c ∈ eList StdLib Split (conc (GetCoveredExtension (eList i) (i ∈ inds eList)) |
  guard : [IUmlInteractionConstraint] = if ∃ gu ∈ eList · gu.name = "guard" then GetGuard (let g ∈ {p | p ∈ eList · p.name = "guard"}) in
  else nil in

return new UmlInteractionOperand (name, fragments, covered, guard)

pre e.name = "operand"

functions
private

GetCoveredExtension : XmlEntity → char*
GetCoveredExtension (e) ∆
  let eList = e.entities.entities in
  let p ∈ {co.data.data | co ∈ eList · co.name = "covered"} in
  p
  pre hasAttributeValue (e, "extender", "umltrans") ∧
  ∃ el ∈ eList e.entities.entities · el.data ≠ nil ∧ el.name = "covered" ;

private

GetCovered : XmlEntity → String*
GetCovered (e) ∆
  let text : String = getAttribute (e, "covered").val in
  StdLib Split (text, ")
  pre hasAttribute (e, "covered") ;
GetGuard : XmlEntity → IUmlInteractionConstraint
GetGuard (e) △
  let elist = e.entities.entities,
  minint = let tmp ∈ \{getGuardConstraintValue (p) | p ∈ elems elist · p.name = "minint"\} in
  maxint = let tmp ∈ \{getGuardConstraintValue (p) | p ∈ elems elist · p.name = "maxint"\} in
  new IUmlInteractionConstraint (minint, maxint)
pre e.name = "guard" ;

private
getGuardConstraintValue : XmlEntity → [IUmlValueSpecification]
getGuardConstraintValue (e) △
  if hasAttribute (e, "value") ∧ isAttributeType (e, "uml:LiteralInteger")
  then new UmlLiteralInteger (StdLib 'StringToInt (getAttribute (e, "value").val))
  else nil
operations
private
buildCallEventMap : XmlEntity-set o String m IUmlCallEvent
buildCallEventMap (elist) △
  let m = \{getAttribute (e, "xmi:id").val ↦ new UmlCallEvent (idToOperationMap (getAttribute (e, "operation").val)) | e ∈ elist\} in
  return m
pre ∀ e ∈ elist · isAttributeType (e, "uml : CallEvent") ;
public
VisitXmlAttribute : XmlAttribute o ()
VisitXmlAttribute (_) △ skip;
public
VisitXmlData : XmlData o ()
VisitXmlData (_) △ skip
end Xml2UmlModel
Test Suite : vdm.tc
Class : Xml2UmlModel

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xml2UmlModel'GetGuard</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>Name</td>
<td>#Calls</td>
<td>Coverage</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Bes</td>
<td>8</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Mos</td>
<td>12</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'GetCovered</td>
<td>23</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'lookUpType</td>
<td>32</td>
<td>88%</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Combi</td>
<td>3</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'VisitXmlData</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Xml2UmlModel'getAttribute</td>
<td>572</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'hasAttribute</td>
<td>220</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'GetVisibility</td>
<td>33</td>
<td>88%</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Message</td>
<td>6</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Operand</td>
<td>4</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'extractClass</td>
<td>15</td>
<td>98%</td>
</tr>
<tr>
<td>Xml2UmlModel'VisitXmlEntity</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Fragment</td>
<td>23</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Property</td>
<td>32</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-defBlock</td>
<td>15</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'getElementType</td>
<td>23</td>
<td>85%</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Operation</td>
<td>5</td>
<td>77%</td>
</tr>
<tr>
<td>Xml2UmlModel'setDefaultValue</td>
<td>32</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'getMultiplicity</td>
<td>32</td>
<td>52%</td>
</tr>
<tr>
<td>Xml2UmlModel'isAttributeType</td>
<td>715</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'VisitXmlDocument</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Constraint</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'VisitXmlAttribute</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Xml2UmlModel'buildCallEventMap</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Association</td>
<td>6</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Interaction</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'hasAttributeValue</td>
<td>739</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'build-Collobration</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'GetCoveredExtension</td>
<td>4</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'GetIntOperationKind</td>
<td>3</td>
<td>√</td>
</tr>
<tr>
<td>Xml2UmlModel'getGuardConstraintValue</td>
<td>4</td>
<td>√</td>
</tr>
</tbody>
</table>

**Total Coverage** 92%
F.2. Transform UML to VDM (Uml2Vdm))

Transform a UML AST into a OML AST.

*init* Convert UML model to OmlDocument.

*build_classes* Build Oml Classes from the Uml model. This needs the constraints and associations to be pre parsed before this is called.

*build_class* Build Oml class from Uml class.

*build_defs* Build definitions from Uml definition block. Here both values and instance variables are extracted.

*build_defValues* Build value definitions from Uml properties.

*build_value* Build value definition from a Uml property.

*build_defInstanceVariables* Build instance variable definition form Uml property.

*build_instanceVariable* Create instance variable from Uml property.

*build_defOperations* Build operation definition from Uml property.

*build_Operation* Create operation from Uml property.

*extractInstanceVarsFromAssociations* extract instance variables form associations.

*hasXorConstraint* Test of an association has a constraint.

*extractBinaryAssociation* extract property from binary association.

*extractUnionAssociation* extract union type from association.

*extractProductAssociation* extract product type from association.

*CreateProductType* Create product type from Uml type.

*CreateInstanceVar* Create instance variable form Uml property and Oml type.

*AddInstanceVarToClass* Add instance variable to the owning class.

*getAssociationInstanceVars* Get instance variables for a specific class extracted from associations.

*getDefaultExpression* Get default expression from Uml default value and type.

*ConvertVisibility* Convert visibility to scope.

*ConvertType* Convert type and multiplicity to Omiltype.
ApplyMultiplicity add multiplicity extracted information to type. eg chat with upper* is a set of char.

build_traces Build traces from interaction.

build_trace build a trace definition from a SD by using the name of the SD as the name of the trace.

getTraceDefinition Get definitions loops through the messages of the SD. It keeps looping until none of the messages are back. It handles one message at the time and groups the according to the fragments they are contained in by the interaction operand.

getLoopDef Get loop definition converts a combined fragment into a Trace definition item where the repeat pattern is set according to the constraint of the interaction operand inside the loop where the messages that is used as test is located.

getRpEx Get repeat pattern from an interaction operand the operand is associated with a message the there by the link to the trace repeat pattern with a test of the message.

getAltDef Get alt definition converts a CF kind of alt into a Choice Definition. Adding the message that is located inside it by the interaction operands.

getMethodApply Get Method Apply from a Message.

getOperand Get Interaction Operand where a message is placed if the message is placed in a IO else return nil.

getCfIoKind Get Combined fragment kind from a set of CF and a InteractionOperand.

class Uml2 Vdm

  types
    public String = char*

  instance variables
    classInstanceVars : String m IOmlInstanceVariable* := {→};

  operations
    public
      init : IUmlModel ▽ IOmlDocument
      init (model) ▽
      let associations = { a | a ∈ model.getDefinitions () ·
        isofclass (IUmlAssociation, a)},
constraints = { a | a ∈ model.getDefinitions () ·
  isofclass ( IUmlConstraint , a ) } in
( extractInstanceVarsFromAssociations ( associations , constraints ) ;
  return new OmlDocument ( model.getName () ,
    new OmlSpecifications ( build-classes ( model ) , [] ) ) )
);

public
build-classes : IUmlModel → IOmlClass*
build-classes ( model ) △
let traceDefMap : String → IOmlTraceDefinitions = build-traces ( model ) ,
classes : IUmlClass-set = { c | c ∈ model.getDefinitions () ·
  isofclass ( IUmlClass , c ) } in
return Util' SetToSeq [ IOmlClass ] ( { let cName = c.getName () ,
  traceDef =
    if cName ∈ dom traceDefMap
    then traceDefMap ( cName )
    else nil in
    build-class ( c , traceDef ) |
    c ∈ classes } ) ;

public
build-class : IUmlClass × [ IOmlTraceDefinitions ] → IOmlClass
build-class ( c , traceDef ) △
let name : char* = c.getName () ,
supers = c.getSuperClass () ,
inheritanceClause : [ OmlInheritanceClause ] =
  if len supers > 0
  then new OmlInheritanceClause ( [ supers ( i ).getName () | i ∈
  inds supers ] )
  else nil ,
body : IOmlDefinitionBlock-set = ∪ { build-defs ( d ) | d ∈ c.getClassBody () } ,
bodyWithTrace : IOmlDefinitionBlock-set = if traceDef ≠ nil
  then body ∪ { traceDef }
  else body ,
systemSpec : B = false ,
instVars : IOmlInstanceVariableShape* = getAssociationInstanceVars ( name ),
APPENDIX F. MODEL COVERAGE

```java
bodyLst = if len instVars > 0
    then bodyWithTrace∪{new OmlInstanceVariableDefinitions (instVars)}
    else bodyWithTrace in
return new OmlClass (name, []
    inheritanceClause,
    Util'SetToSeq[IOmlDefinitionBlock] (bodyLst),
    systemSpec);

public
build-defs : IUmlDefinitionBlock → IOmlDefinitionBlock-set
build-defs (db) △
cases true:
    (isofclass (IUmlOwnedProperties, db)) →
        let tmp : IUmlOwnedProperties = db in
        return {build-defValues (tmp)} ∪
        {build-defInstanceVariables (tmp)},
    (isofclass (IUmlOwnedOperations, db)) →
        let tmp : IUmlOwnedOperations = db in
        return {build-defOperations (tmp)},
    others → return {}
end;

public
build-defValues : IUmlOwnedProperties → IOmlDefinitionBlock
build-defValues (block) △
let props = block.getPropetyList (),
    valueProps = {p | p ∈ props ·
        p.hasIsReadOnly () ∧ p.getIsReadOnly () =
        true},
    val = {build-value (v) | v ∈ valueProps} in
return new OmlValueDefinitions (Util'SetToSeq[IOmlValueDefinition] (val));

public
build-value : IUmlProperty → IOmlValueDefinition
build-value (prop) △
let asyncAccess = false,
    statAccess = prop.getIsStatic (),
    scope = ConvertVisibility (prop.getVisibility ()),
    access = new OmlAccessDefinition (asyncAccess, statAccess, scope),
    pattern = new OmlPatternIdentifier (prop.getName ()),
    multiplicity = if prop.hasMultiplicity ()
        then prop.getMultiplicity ()
        else nil ;
```
type = ConvertType (prop.getType (), multiplicity),
expression : IOmlExpression =
  getDefaultExpression (prop.getDefault (), type),
valueShape = new OmlValueShape (pattern, type, expression) in
return new OmlValueDefinition (access, valueShape)

public
build-defInstanceVariables : IUmlOwnedProperties \(\rightarrow\) IOmlDefinitionBlock
build-defInstanceVariables (block) \(\triangle\)
let props = block.getPropertyList (),
valueProps = \{ p \mid p \in props ·
  p.hasIsReadOnly () \land
  p.getIsReadonly () = false \},
val = \{ build-instanceVariable (v) \mid v \in valueProps \},
seqVal = Util.SetToSeq [IOmlInstanceVariable] (val) in
return new OmlInstanceVariableDefinitions (seqVal);

public
build-instanceVariable : IUmlProperty \(\rightarrow\) IOmlInstanceVariable
build-instanceVariable (prop) \(\triangle\)
let asyncAccess = false,
statAccess = prop.getIsStatic (),
scope = ConvertVisibility (prop.getVisibility ()),
access = new OmlAccessDefinition (asyncAccess, statAccess, scope),
multiplicity = if prop.hasMultiplicity ()
  then prop.getMultiplicity ()
  else nil ,
type = ConvertType (prop.getType (), multiplicity),
expression : [IOmlExpression] =
  if prop.hasDefault ()
  then getDefaultExpression (prop.getDefault (), type)
  else nil ,
assignmentDef =
  new OmlAssignmentDefinition (prop.getName (), type, expression) in
return new OmlInstanceVariable (access, assignmentDef);

public
build-defOperations : IUmlOwnedOperations \(\rightarrow\) IOmlDefinitionBlock
build-defOperations (block) \(\triangle\)
let props = block.getOperationList (),
valOps = \{ p \mid p \in props \},
val = \{ build-Operation (v) \mid v \in valOps \},
seqVal = Util.SetToSeq[IOmlOperationDefinition](val)
return new OmlOperationDefinitions(seqVal);

public build-Operation : IUmlOperation → IOmlOperationDefinition
build-Operation(prop) △
let asyncAccess = false,
statAccess = prop.getIsStatic(),
scope = ConvertVisibility(prop.getVisibility()),
access = new OmlAccessDefinition(asyncAccess, statAccess, scope),
type = new OmlOperationType(new OmlEmptyType(), new OmlEmptyType()),
name : char* =
prop.getName(),
body = new OmlOperationBody(new OmlSkipStatement(), false, false),
trailer = new OmlOperationTrailer(nil, nil, nil, nil),
explicitOp = new OmlExplicitOperation(name, type, [], body, trailer) in
return new OmlOperationDefinition(access, explicitOp);

public extractInstanceVarsFromAssociations : IUmlAssociation-set × IUmlConstraint-set → ()
extractInstanceVarsFromAssociations(associations, constraints) △
let normalBiAss =
{ a | a ∈ associations ·
  ¬hasXorConstraint(constraints, a.getId()) ∧
  (card a.getOwnedEnds() + card a.getOwnedNavigableEnds()) = 2 },
product =
{ a | a ∈ associations ·
  ¬hasXorConstraint(constraints, a.getId()) ∧
  (card a.getOwnedEnds() + card a.getOwnedNavigableEnds()) > 2 },
xor =
{ a | a ∈ associations ·
  hasXorConstraint(constraints, a.getId()) ∧
  (card a.getOwnedEnds() + card a.getOwnedNavigableEnds()) ≥ 2 }
in
( for all a ∈ normalBiAss }
do extractBinaryAssociation
(a.getOwnedEnds () ∪
a.getOwnedNavigableEnds ()) :
let xorEndss = {
(( ∪ {a.getOwnedEnds () ∪
a.getOwnedNavigableEnds () |
a ∈ xor .
∃ id ∈ c.getConstraintElements () ·
id = a.getId ()}) |
c ∈ constraints)} in
for all a ∈ xorEndss
do extractUnionAssociation (a) :
for all a ∈ product
do extractProductAssociation
(a.getOwnedEnds () ∪
a.getOwnedNavigableEnds ())
);

c : IUmlClassNameType = pOwnerEnd .
len propSeq (i) .
if pTypeEnd .
hasXorConstraint : IUmlConstraint-set × String → 
hasXorConstraint (constraints, associationId) △
return ∃ c ∈ constraints ·
if isofclass (IUmlLiteralString, c.getSpecification ())
then let spec : IUmlLiteralString = c.getSpecification () in
spec.getValue () = "xor"
else false ∧
(∃ ce ∈ c.getConstraintElements () ·
ce = associationId);

e xtractBinaryAssociation : IUmlProperty-set → ()
extractBinaryAssociation (props) △
( let propSeq = Util.SetToSeq[IUmlProperty] (props),
  pOwnerEnd = hd [propSeq (i) | i ∈ inds propSeq ·
  len propSeq (i) .
  if pTypeEnd .
multiplicity = if pTypeEnd .
```
type = ConvertType (pTypeEnd.getType (), multiplicity) in
AddInstanceVarToClass (clName,
    CreateInstanceVar (pTypeEnd,
        type))
)

pre card props > 0 ;
pubic

extractUnionAssociation : IUmlProperty\-set \rightarrow ()
extractUnionAssociation (props) \triangleq
(  let ownerEndSet = {p | p \in props \cdot len p.getName () = 0},
    propSeq = Util\SetToSeq [IUmlProperty] (props),
    pOwnerEnd = hd Util\SetToSeq [IUmlProperty] (ownerEndSet),
    pTypeEnd = [propSeq (i) | i \in inds propSeq \cdot
        len propSeq (i).getName () > 0],
    clName = let t : IUmlClassNameType = pOwnerEnd.getType () in
        t.getName (),
    endTypes : IOmlType\* =
        Util\SetToSeq [IOmlType] (\{ ConvertType (p.getType (), if p.hasMultiplicity ()
        then p.getMultiplicity ()
        else nil ) | p \in elems pTypeEnd \}),
    lhs : IOmlType = hd endTypes,
    rhs : IOmlType = hd Util\SetToSeq [IOmlType] ((elems endTypes) \{lhs\}),
)
    type = new OmlUnionType (lhs, rhs) in
AddInstanceVarToClass (clName, CreateInstanceVar (hd pTypeEnd, type))
)

pre card props > 0 ;
pubic

extractProductAssociation : IUmlProperty\-set \rightarrow ()
extractProductAssociation (props) \triangleq
(  let ownerEndSet = {p | p \in props \cdot len p.getName () = 0},
    propSeq = Util\SetToSeq [IUmlProperty] (props),
    pOwnerEnd = hd Util\SetToSeq [IUmlProperty] (ownerEndSet),
    pTypeEnd = [propSeq (i) | i \in inds propSeq \cdot
        len propSeq (i).getName () > 0],
    clName = let t : IUmlClassNameType = pOwnerEnd.getType () in
        t.getName (),
    endTypes : IOmlType\* =
        Util\SetToSeq [IOmlType] (\{ ConvertType (p.getType (), if p.hasMultiplicity ()
        then p.getMultiplicity ()
        else nil ) | p \in elems pTypeEnd \}),
    lhs : IOmlType = hd endTypes,
    rhs : IOmlType = hd Util\SetToSeq [IOmlType] ((elems endTypes) \{lhs\}),
    type = new OmlUnionType (lhs, rhs) in
AddInstanceVarToClass (clName, CreateInstanceVar (hd pTypeEnd, type))
)```
endTypes : IUmlType* = Util.SetToSeq(IUmlType) ({ p.getType() | p \in elems pTypeEnd }),
type : IOmlType = CreateProductType (endTypes) in
AddInstanceVarToClass(clName, CreateInstanceVar (hd pTypeEnd, type))
)  
pre card props > 0

functions
private
CreateProductType : IUmlType* \rightarrow IOmlType
CreateProductType (tps) \triangleq
let first = hd tps,
    rest = tl tps,
    front = ConvertType (first, nil) in
    if len tps = 1
    then front
    else new OmlProductType (front, CreateProductType (rest))

operations
public
CreateInstanceVar : IUmlProperty \times IOmlType \rightarrow IOmlInstanceVariable
CreateInstanceVar (prop, type) \triangleq
let asyncAccess =
    false,
    statAccess = prop.getIsStatic (),
    scope = ConvertVisibility (prop.getVisibility ()),
    access = new OmlAccessDefinition (asyncAccess, statAccess, scope),
    multiplicity = if prop.hasMultiplicity ()
        then prop.getMultiplicity ()
        else nil ,
    type1 = ConvertType (prop.getType (), multiplicity),
    expression : [IOmlExpression] =
        if prop.hasDefault ()
            then getDefaultExpression (prop.getDefault (), type1)
            else nil ,
    assignmentDef = new OmlAssignmentDefinition (prop.getName (), type, expression) in
    return new OmlInstanceVariable (access, assignmentDef);

private
AddInstanceVarToClass : String \times IOmlInstanceVariable \rightarrow ()
AddInstanceVarToClass (clName, instanceVar) \triangleq
let existingSet = getAssociationInstanceVars (clName),
\[
\text{addedSet} = \begin{cases} 
\text{len existingSet} > 0 \\
\emptyset 
\end{cases}
\]
\[
\text{then existingSet} \rightarrow \{\text{instanceVar}\} \\
\text{else} \{\text{instanceVar}\} \text{ in} \\
\text{classInstanceVars} := \text{classInstanceVars} \uplus \{\text{clName} \mapsto \text{addedSet}\};
\]

**public**

\[
\text{getAssociationInstanceVars} : \text{String} \rightarrow \text{IOmlInstanceVariableShape}^* \\
\text{getAssociationInstanceVars} (\text{clName}) \triangleq \\
\text{if clName} \in \text{dom classInstanceVars} \\
\text{then return classInstanceVars (clName)} \\
\text{else return} \emptyset ;
\]

**public**

\[
\text{getDefaultExpression} : [\text{char}] \times \text{IOmlType} \rightarrow [\text{IOmlExpression}] \\
\text{getDefaultExpression} (\text{defaultValue}, \text{t}) \triangleq \\
\text{if defaultValue} = \text{nil} \\
\text{then return nil} \\
\text{else cases true:} \\
\text{(isofclass} (\text{IOmlTypeName}, \text{t})) \rightarrow \\
\text{return new OmlNewExpression (new OmlName (nil, defaultValue), \emptyset, \emptyset);} \\
\text{(isofclass} (\text{IOmlIntType}, \text{t})) \rightarrow \\
\text{return new OmlSymbolicLiteralExpression (new OmlNumericLiteral (0));} \\
\text{(isofclass} (\text{IOmlCharType}, \text{t})) \rightarrow \\
\text{return new OmlSymbolicLiteralExpression (new OmlTextLiteral (defaultValue));} \\
\text{(isofclass} (\text{IOmlSeq0Type}, \text{t})) \rightarrow \\
\text{return getDefaultExpression (defaultValue, let tmp:IOmlSeq0Type in} \\
\text{tmp.getType());} \\
\text{others} \rightarrow \text{return nil}
\]

**public**

\[
\text{ConvertVisibility} : \text{IUmlVisibilityKind} \rightarrow \text{IOmlScope} \\
\text{ConvertVisibility} (\text{visibility}) \triangleq \\
\text{let val} : \text{N} = \text{visibility.getValue()} \text{ in} \\
\text{cases val:} \\
(\text{UmlVisibilityKindQuotes'IQPUBLIC}) \rightarrow \\
\text{return new OmlScope (OmlScopeQuotes'IQPUBLIC);} \\
(\text{UmlVisibilityKindQuotes'IQPRIVATE}) \rightarrow \\
\text{return new OmlScope (OmlScopeQuotes'IQDEFAULT);} \\
(\text{UmlVisibilityKindQuotes'IQPROTECTED}) \rightarrow \\
\text{return new OmlScope (OmlScopeQuotes'IQPROTECTED);}
\]

**public**
F.2. TRANSFORMING UML TO VDM

\[\text{ConvertType} : IUmlType \times [IUmlMultiplicityElement] \triangleright IOmlType\]
\[\text{ConvertType} (t, \text{mul}) \triangleq\]
cases true:
\(\text{isofclass} (IUmlClassNameType, t)) \rightarrow\)
\(\text{return new OmlTypeName (let tmp : IUmlClassNameType = t in}\)
\(\text{new OmlName (nil, tmp.getName ())}).\)
\(\text{isofclass} (IUmlCharType, t)) \rightarrow \text{return ApplyMultiplicity (new OmlCharType (), mul),}\)
\(\text{isofclass} (IUmlStringType, t)) \rightarrow \text{return new OmlSeq0Type (new OmlCharType ())},\)
\(\text{isofclass} (IUmlIntegerType, t)) \rightarrow \text{return ApplyMultiplicity (new OmlIntType (), mul)},\)
\(\text{isofclass} (IUmlBoolType, t)) \rightarrow \text{return ApplyMultiplicity (new OmlBoolType (), mul)},\)
\(\text{isofclass} (IUmlUnlimitedNatural, t)) \rightarrow \text{return ApplyMultiplicity (new OmlRealType (), mul),}\)
others \rightarrow \text{return ApplyMultiplicity (new OmlNatType (), mul)}\)
end;

private
\[\text{ApplyMultiplicity} : IOmlType \times [IUmlMultiplicityElement] \triangleright IOmlType\]
\[\text{ApplyMultiplicity} (t, \text{mul}) \triangleq\]
if \(\text{mul} = \text{nil}\)
then return \(t\)
else if \(\text{mul}.\text{getLower} () = 0 \land \neg \text{mul}.\text{hasUpper} ()\)
then return \(\text{new OmlSeq0Type (t)}\)
else return \(t\)

functions
private
\[\text{build-traces} : IUmlModel \rightarrow String \triangleright IOmlTraceDefinitions\]
\[\text{build-traces} (\text{model}) \triangleq\]
let \(\text{collOwnedBehavior} = \bigcup \{\text{let tmp : IUmlCollaboration} =\)
\(\text{coll in}\)
\(\text{tmp.getOwnedBehavior ()} \mid\)
\(\text{coll} \in \text{model.getDefinitions ()} \cdot\)
\(\text{isofclass} (IUmlCollaboration, \text{coll})\}\}
in
merge \{\text{build-trace} (interaction) \mid \text{interaction} \in \text{collOwnedBehavior}\};
build-trace : IUmlInteraction $\rightarrow$ String $\rightarrow$ IOmlTraceDefinitions

build-trace (interaction) $\triangleq$
    let name = interaction.getName (),
        messages : IUmlMessage* = interaction.getMessages () in
    let defs : IOmlTraceDefinition =
        getTraceDefinition (messages, interaction.getFragments (), nil ) in
    let ownerClass $\in\{m.getSendEvent ().getCovered ().getRepresents () |
        m $\in$ elems messages\} in
    {let our : IUmlClassNameType = ownerClass in
        our.getName () $\mapsto$
        new OmlTraceDefinitions ([new OmlNamedTrace (name, defs)])};
F.2. TRANSFORMING UML TO VDM

\[ \text{getTraceDefinition} : IUmlMessage}^* \times IUmlInteractionFragment \rightarrow \]
\[ [IUmlInteractionOperand] \rightarrow [IOmlTraceDefinition] \]
\[ \text{getTraceDefinition} (msgs, fg, io) \triangleq \]
\[ \begin{cases} \text{let } m = \text{hd } msgs, \\
\quad \text{rest } = \text{if len } msgs > 1 \\
\quad \text{then tl } msgs \\
\quad \text{else } [], \end{cases} \]
\[ \begin{aligned} \text{cfg } &= \{ f \mid f \in fg \cdot \text{isofclass} (IUmlCombinedFragment, f) \}, \\
\text{op } &= \text{getOperand} (m, \text{cfg}) \text{ in} \end{aligned} \]
\[ \begin{cases} \text{if } (\text{io } = \text{nil } \land \text{op } = \text{nil }) \lor (\text{io } = \text{op}) \\
\text{then let } \text{mappDef } : \text{IOmlTraceDefinition } = \end{cases} \]
\[ \begin{aligned} \text{new OmlTraceDefinitionItem } ([], \text{getMethodApply} (m), \text{nil } ), \\
\text{restDef } : \text{IOmlTraceDefinition } = \text{getTraceDefinition} (\text{rest}, \text{cfg}, \text{op}), \\
\text{defs } : \text{IOmlTraceDefinition}^* &= [\text{mappDef}, \text{restDef}] \text{ in} \end{aligned} \]
\[ \begin{aligned} \text{let ret } : \text{IOmlTraceDefinition } = \text{new OmlTraceSequenceDefinition } (\text{defs}) \text{ in} \end{aligned} \]
\[ \text{ret } \\
\text{else if } \text{op } \neq \text{nil } \land \text{getCfIoKind} (fg, \text{op}).\text{getValue} () = \]
\[ \text{UmlInteractionOperatorKindQuotes IQLOOP} \]
\[ \text{then let loopDef } : \text{IOmlTraceDefinition } = \text{getLoopDef} (m, \text{op}), \]
\[ \text{restDef } : \text{IOmlTraceDefinition } = \text{getTraceDefinition} (\text{rest}, \text{cfg}, \text{op}), \\
\text{defs } : \text{IOmlTraceDefinition}^* &= \begin{cases} \text{if restDef } \neq \text{nil} \\
\text{then } [\text{loopDef}, \text{restDef}] \\
\text{else } [\text{loopDef}], \end{cases} \text{ in} \]
\[ \begin{aligned} \text{let ret } : \text{IOmlTraceDefinition } = \text{new OmlTraceSequenceDefinition } (\text{defs}) \text{ in} \end{aligned} \]
\[ \text{ret } \\
\text{else if } \text{op } \neq \text{nil } \land \text{getCfIoKind} (fg, \text{op}).\text{getValue} () = \]
\[ \text{UmlInteractionOperatorKindQuotes IQALT} \]
\[ \text{then let altDef } : \text{IOmlTraceDefinition } = \text{getAltDef} (\text{rest}, m, \text{cfg}, \text{op}), \\
\text{restDef } : \text{IOmlTraceDefinition } = \text{getTraceDefinition} (\text{rest}, \text{cfg}, \text{op}), \\
\text{defs } : \text{IOmlTraceDefinition}^* &= \begin{cases} \text{if restDef } \neq \text{nil} \\
\text{then } [\text{altDef}, \text{restDef}] \\
\text{else } [\text{altDef}], \end{cases} \text{ in} \]
\[ \begin{aligned} \text{let ret } : \text{IOmlTraceDefinition } = \text{new OmlTraceSequenceDefinition } (\text{defs}) \text{ in} \end{aligned} \]
\[ \text{ret } \\
\text{else let ret } : \text{IOmlTraceDefinition } = \text{new OmlTraceSequenceDefinition } () \end{cases} \text{ in} \]
\[ \text{ret}) \}
\[ \text{else nil ;} \]
getLoopDef : IUmlMessage × [IUmlInteractionOperand] → IOmlTraceDefinition
getLoopDef (m, io) △
  new OmlTraceDefinitionItem ([],
  getMethodApply (m),
  getRpEx (io));

private

getRpEx : [IUmlInteractionOperand] → [IOmlTraceRepeatPattern]
getRpEx (iOperand) △
  if iOperand = nil
  then nil
  else if iOperand.hasGuard ()
    then let guard = iOperand.getGuard (),
       min = if guard.hasMinint ()
         then let tmp : IUmlLiteralInteger =
           guard.getMinint () in
           tmp.getValue ()
         else nil,
       max = if guard.hasMaxint ()
         then let tmp : IUmlLiteralInteger =
           guard.getMaxint () in
           tmp.getValue ()
         else nil in
       if min ≠ nil ∧ min = 0 ∧ max = nil
         then new OmlTraceZeroOrMore ()
       else if min ≠ nil ∧ min = 1 ∧ max = nil
         then new OmlTraceOneOrMore ()
       else if min ≠ nil ∧ max ≠ nil ∧ min = 0 ∧ max = 1
         then new OmlTraceZeroOrOne ()
       else if min ≠ nil
         then let minL = new OmlNumericLiteral (min),
            maxL = if max ≠ nil
              then new OmlNumericLiteral (max)
            else nil in
            new OmlTraceRange (minL, maxL)
       else nil
  else nil;

private
getAltDef : IUmlMessage * IUmlMessage * IUmlCombinedFragment-set * [IUmlInteractionOperand] → IOmlTraceDefinition
getAltDef (msgs, m, fg, io) △
  let mapp = getMethodApply (m),
      rest = getTraceDefinition (msgs, fg, io),
      defs = if rest ≠ nil
        then [new OmlTraceDefinitionItem ([], mapp, nil), rest]
        else [new OmlTraceDefinitionItem ([], mapp, nil)]
      in
    new OmlTraceChoiceDefinition (defs);

private
getMethodApply : IUmlMessage → IOmlTraceMethodApply
getMethodApply (message) △
  let methodName : String =
    message.getSendReceive ().getEvent ().getOperation ().getName (),
  variableName : String =
    message.getSendReceive ().getCovered ().getName (),
  args : IOmlExpression * = []
  in
    new OmlTraceMethodApply (variableName, methodName, args);

private
getOperand : IUmlMessage * IUmlCombinedFragment-set → [IUmlInteractionOperand]
getOperand (m, fragments) △
  let ops = { io | io ∈ elems conc Util'stToSeq [IUmlInteractionOperand *]
    (   { f.getOperand () | f ∈ fragments ·
        isofclass (IUmlCombinedFragment, f)})·
    ∃ mos ∈ io.getCovered () ·
    mos = m.getSendEvent () } in
  if card ops > 0
    then let p ∈ ops in
      p
    else nil ;

private
getCfIoKind : IUmlCombinedFragment $\rightarrow$ IUmlInteractionOperatorKind

getcfIoKind (fms, io) $\triangleq$

\[
\text{let } cf \in \\
\{ f | f \in fms \cdot \\
\quad \text{isofclass} (IUmlCombinedFragment, f) \land \\
\quad \exists iop \in \text{elems } f \cdot \text{getOperand}() \cdot iop = io \} \\
\text{in } cf \cdot \text{getInteractionOperator}()
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uml2Vdm'init</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'getRpEx</td>
<td>2</td>
<td>85%</td>
</tr>
<tr>
<td>Uml2Vdm'getAltDef</td>
<td>3</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'build-defs</td>
<td>30</td>
<td>92%</td>
</tr>
<tr>
<td>Uml2Vdm'getLoopDef</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'getOperand</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'ConvertType</td>
<td>26</td>
<td>70%</td>
</tr>
<tr>
<td>Uml2Vdm'build-class</td>
<td>15</td>
<td>96%</td>
</tr>
<tr>
<td>Uml2Vdm'build-trace</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'build-value</td>
<td>1</td>
<td>92%</td>
</tr>
<tr>
<td>Uml2Vdm'build-traces</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'build-classes</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'getMethodApply</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'build-Operation</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'build-defValues</td>
<td>15</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'hasXorConstraint</td>
<td>18</td>
<td>62%</td>
</tr>
<tr>
<td>Uml2Vdm'ApplyMultiplicity</td>
<td>14</td>
<td>27%</td>
</tr>
<tr>
<td>Uml2Vdm'ConvertVisibility</td>
<td>23</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'CreateInstanceVar</td>
<td>4</td>
<td>78%</td>
</tr>
<tr>
<td>Uml2Vdm'CreateProductType</td>
<td>3</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'getTraceDefinition</td>
<td>12</td>
<td>96%</td>
</tr>
<tr>
<td>Uml2Vdm'build-defOperations</td>
<td>15</td>
<td>√</td>
</tr>
<tr>
<td>Uml2Vdm'getDefaultExpression</td>
<td>5</td>
<td>51%</td>
</tr>
<tr>
<td>Uml2Vdm'AddInstanceVarToClass</td>
<td>4</td>
<td>80%</td>
</tr>
<tr>
<td>Uml2Vdm'build-instanceVariable</td>
<td>13</td>
<td>92%</td>
</tr>
<tr>
<td>Uml2Vdm'extractUnionAssociation</td>
<td>1</td>
<td>96%</td>
</tr>
<tr>
<td>Uml2Vdm'extractBinaryAssociation</td>
<td>2</td>
<td>95%</td>
</tr>
<tr>
<td>Name</td>
<td>#Calls</td>
<td>Coverage</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>Uml2Vdm’extractProductAssociation</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2Vdm’build-defInstanceVariables</td>
<td>15</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2Vdm’getAssociationInstanceVars</td>
<td>19</td>
<td>✓</td>
</tr>
<tr>
<td>Uml2Vdm’extractInstanceVarsFromAssociations</td>
<td>7</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Total Coverage</strong></td>
<td></td>
<td><strong>91%</strong></td>
</tr>
</tbody>
</table>
F.3 OML AST to VDM files printer

F.3.1 Proxy for printer (Oml2Vpp)

Provides a easy to use interface for printing a OML AST to source files. Connects the Oml2Vpp visitor to the IO facility.

class Oml2Vpp

types

        public String = char*

operations

g public

        Save : String × IOmlDocument → ()

        Save (fileName, doc) △

        (        dcl visitor : Oml2VppVisitor := new Oml2VppVisitor ();

        visitor.visitDocument(doc);

        Util'SetName(fileName);

        Util'PrintL(visitor.result)

)

end Oml2Vpp

Test Suite : vdm.tc

Class : Oml2Vpp

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oml2VppSave</td>
<td>?</td>
<td>✓</td>
</tr>
<tr>
<td>Total Coverage</td>
<td></td>
<td>100%</td>
</tr>
</tbody>
</table>
F.3.2 Visitor for OML which implements a printer for source files
(OMl2VppVisitor)

Implementation of the OML source file printer.

class Oml2VppVisitor is subclass of OmlVisitor

types

\[ String = char^* \]

values

private

\( nl : char^* ="\n" \)

instance variables

public result : char* := [];

private lvl : N := 0;

operations

private

\[ printNodeField : IOmlNode \to () \]

\[ printNodeField (pNode) \triangleq pNode. \]

accept (self);

private

\[ printBoolField : B \to () \]

\[ printBoolField (pval) \triangleq \]

\[ result := if \ pval \]

\[ then "true" \]

\[ else "false" ; \]

private

\[ printNatField : N \to () \]

\[ printNatField (pval) \triangleq \]

\[ result := StdLib' ToStringInt (pval) ; \]

private

\[ printRealField : R \to () \]

\[ printRealField (-) \triangleq \]

\[ error ; \]

private

\[ printCharField : char \to () \]

\[ printCharField (pval) \triangleq \]

\[ result := "\n" \to[pval] \to "\n" ; \]

private

\[ printField : IOmlNode' FieldValue \to () \]

\[ printField (fld) \triangleq \]

\[ if is-B (fld) \]

\[ then printBoolField (fld) \]
else if is-char (fld)
then printCharField (fld)
else if is-N (fld)
then printNatField (fld)
else if is-R (fld)
then printRealField (fld)
else isofclass (IOmlNode, fld)
then printNodeField (fld)
else print StringField (fld); 

private
printlnStringField : char\* \xrightarrow{} ()
printlnStringField (str) \triangleq
result := "\n" \xrightarrow{} str \xrightarrow{} "\n";

private
printlnSeqofField : IOmlNode'FieldValue\* \xrightarrow{} ()
printlnSeqofField (pval) \triangleq
(dcl str : char\* := " ",
cnt : N := len pval;
while cnt > 0
do ( printlnField(pval(len pval − cnt + 1));
str := str \xrightarrow{} result;
cnt := cnt - 1
);
result := str
);

public
visitNode : IOmlNode \xrightarrow{} ()
visitNode (pNode) \triangleq pNode.
accept(self);

public
visitDocument : IOmlDocument \xrightarrow{} ()
visitDocument (pcmp) \triangleq
(dcl str : char\* := "BEGIN FileName: " \xrightarrow{} pcmp.getFilename () \xrightarrow{} "\n";
if pcmp.hasSpecifications ()
then printlnSpecifications(pcmp.getSpecifications ());
result := str \xrightarrow{} result \xrightarrow{} "END FileName: " \xrightarrow{} pcmp.getFilename ()
);

public
visitSpecifications : IOmlSpecifications → ()
visitSpecifications(pcmp) △
  ( dcl str : char* := nl;
   for node in pcmp.getClassList()
     do ( printNodeField(node);
          str := str ∼ nl ∼ result ∼ nl
        );
   result := str
  );

public
visitClass : IOmlClass → ()
visitClass(pcmp) △
  ( dcl str : char* := "class " ∼ pcmp.getIdentifier();
   if pcmp.hasInheritanceClause() then
     printNodeField(pcmp.getInheritanceClause());
   else result := "";
   if db in pcmp.getClassBody() then
     do ( printNodeField(db);
          str := str ∼ nl ∼ result
        );
   result := str ∼ nl ∼ "end" ∼ pcmp.getIdentifier()
  );

public
visitInheritanceClause : IOmlInheritanceClause → ()
visitInheritanceClause(pcmp) △
  ( dcl str : char* := " is subclass of ",
    list : String* := pcmp.getIdentifierList(),
    length : N := len list,
    i : N := 1;
   while i ≤ length
     do ( str := str ∼ list(i);
          i := i + 1;
          if i ≤ length
            then str := str ∼ " , "
          );
   result := str ∼ nl
  );

public
visitValueDefinitions : IOmlValueDefinitions → ()
visitValueDefinitions(pcmp) △
  ( dcl str : char* := nl ∼ "values" ∼ nl;
for db in pcmp.getValueList ()
do (   printNodeField(db);
    str := str \(\bowtie\) result \(\bowtie\) nl ) ;
  if len pcmp.getValueList () = 0
    then result := ""
  else result := str
); public

visitValueDefinition : IOmlValueDefinition \(\rightarrow\) ()
visitValueDefinition (pcmp) \(\triangleq\)
  (  dcl str : char";
   printNodeField(pcmp.getAccess () ) ;
   str := result;
   printNodeField(pcmp.getShape () ) ;
   str := str \(\bowtie\) result \(\bowtie\) ";" \(\bowtie\) nl;
   result := str
); public

visitAccessDefinition : IOmlAccessDefinition \(\rightarrow\) ()
visitAccessDefinition (pcmp) \(\triangleq\)
  (  dcl str : char" := "";
   if pcmp.getStaticAccess ()
     then str := " static ";
   printNodeField(pcmp.getScope () ) ;
   str := str \(\bowtie\) result \(\bowtie\) " ";
   result := str
); public

visitScope : IOmlScope \(\rightarrow\) ()
visitScope (pNode) \(\triangleq\)
  (  cases pNode.getValue () :
      (OmlScopeQuotes"IQPUBLIC") \(\rightarrow\) result := " public",
      (OmlScopeQuotes"IQPRIVATE") ,
      (OmlScopeQuotes"IQDEFAULT") \(\rightarrow\) result := " private",
      (OmlScopeQuotes"IQPROTECTED") \(\rightarrow\) result := " protected",
      others \(\rightarrow\) error
    end
  );
public
F.3. OML AST TO VDM FILES PRINTER

visit ValueShape : IOmlValueShape → ()
visit ValueShape (pcmp) △
   (   
       dcl str : char*;
       printNodeField(pcmp.getPattern());
       str := result ↞ " ";
       if pcmp.hasType ()
       then (      
           printNodeField(pcmp.getType());
           str := str ↞ " " ↞ " " result ↞ " "
       )
       else result := " ";
       printNodeField(pcmp.getExpression());
       str := str ↞ " " ↞ " " result ↞ " ";
       result := str
   );

public

visit Pattern : IOmlPattern → ()
visit Pattern (pNode) △ pNode.
   accept(self);

public

visit Expression : IOmlExpression → ()
visit Expression (pNode) △ pNode.
   accept(self);

public

visit Literal : IOmlLiteral → ()
visit Literal (pNode) △ pNode.
   accept(self);

public

visit Type : IOmlType → ()
visit Type (pNode) △ pNode.
   accept(self);

public

visit PatternIdentifier : IOmlPatternIdentifier → ()
visit PatternIdentifier (pcmp) △
   (   
       dcl str : char* := pcmp.getIdentifier () ↞ " ";
       result := str
   );
visitSymbolicLiteralExpression : IOmlSymbolicLiteralExpression → ()

visitSymbolicLiteralExpression (pcmp) ≐
  ( printNodeField(pcmp.getLiteral ());
)

public

visitTextLiteral : IOmlTextLiteral → ()
visitTextLiteral (pcmp) ≐
  ( dcl str : char* := pcmp.getVal ();
    result := "\" str "\";
    printNodeField(pcmp.getType ());
    str := str result;
    result := str
  );

public

visitSeq0Type : IOmlSeq0Type → ()
visitSeq0Type (pcmp) ≐
  ( dcl str : char* := "seq of ";
    printNodeField(pcmp.getType ());
    str := str result;
    result := str
  );

public

visitCharType : IOmlCharType → ()
visitCharType (-) ≐
  ( dcl str : char* := "char";
    result := str
  );

public

visitInstanceVariableDefinitions : IOmlInstanceVariableDefinitions → ()

visitInstanceVariableDefinitions (pcmp) ≐
  ( dcl str : char* := nl "instance variables" nl nl;
    for db in pcmp.getVariablesList ()
      do ( printNodeField(db);
          str := str result nl
        );
    if len pcmp.getVariablesList () = 0
      then result := ""
    else result := str
  );

public
visitInstanceVariable : IOmlInstanceVariable \rightarrow ()
visitInstanceVariable (pcmp) \triangleq
  ( dcl str : char* := "";
   printNodeField(pcmp.getAccess ());
   str := str \rightarrow result;
   printNodeField(pcmp.getAssignmentDefinition ());
   str := str \rightarrow result;
   result := str );

public

visitAssignmentDefinition : IOmlAssignmentDefinition \rightarrow ()
visitAssignmentDefinition (pcmp) \triangleq
  ( dcl str : char* := "";
   str := str \rightarrow pcmp.getIdentifier ();
   printNodeField(pcmp.getType ());
   str := str \rightarrow " : " \rightarrow result;
   if pcmp.hasExpression ()
     then ( printNodeField(pcmp.getExpression ());
       str := str \rightarrow " : = "
     )
   else result := "";
   str := str \rightarrow result \rightarrow " ; " ;
   result := str );

public

visitTypeName : IOmlTypeName \rightarrow ()
visitTypeName (pcmp) \triangleq
  ( printNodeField(pcmp.getName ()) );

public

visitName : IOmlName \rightarrow ()
visitName (pcmp) \triangleq
  ( dcl str : char* := "";
    if pcmp.hasClassIdentifier ()
      then str := str \rightarrow pcmp.getClassIdentifier ();
    str := str \rightarrow pcmp.getIdentifier ();
    result := str );

public
APPENDIX F. MODEL COVERAGE

```java
public void visitIntType : IOmlIntType o → ()
visitIntType (-) △
(  dcl str : char* := "int";
    result := str
);

public void visitNatType : IOmlNatType o → ()
visitNatType (-) △
(  dcl str : char* := "nat";
    result := str
);

public void visitNat1Type : IOmlNat1Type o → ()
visitNat1Type (-) △
(  dcl str : char* := "nat1";
    result := str
);

public void visitSeq1Type : IOmlSeq1Type o → ()
visitSeq1Type (pcmp) △
(  dcl str : char* := "seq1 of ";
    printNodeField(pcmp.getType ());
    str := str ◦ result;
    result := str
);

public void visitRealType : IOmlRealType o → ()
visitRealType (-) △
(  dcl str : char* := "real";
    result := str
);

public void visitSetType : IOmlSetType o → ()
visitSetType (pcmp) △
(  dcl str : char* := "set of ";
    printNodeField(pcmp.getType ());
    str := str ◦ result;
    result := str
);
```

visitTypeDefinitions : IOmlTypeDefinitions \xrightarrow{o} ()

visitTypeDefinitions (pcmp) \triangleq
  ( dcl str : char\ast := nl \concat " types" \concat nl \concat nl;
    for db in pcmp.getTypeList ()
    do ( printNodeField(db);
         str := str \concat result \concat nl
    )
    result := str
);

public

visitTypeDefinition : IOmlTypeDefinition \xrightarrow{o} ()

visitTypeDefinition (pcmp) \triangleq
  ( dcl str : char\ast := " ";
    printNodeField(pcmp.getAccess ());
    str := str \concat result;
    printNodeField(pcmp.getShape ());
    str := str \concat result \concat ". ";
    result := str
);

public

visitSimpleType : IOmlSimpleType \xrightarrow{o} ()

visitSimpleType (pcmp) \triangleq
  ( dcl str : char\ast := pcmp.getIdentifier ();
    printNodeField(pcmp.getType ());
    result := str \concat " = " \concat result
);

public

visitEmptyType : IOmlEmptyType \xrightarrow{o} ()

visitEmptyType () \triangleq
  ( dcl str : char\ast := " ()";
    result := str
);

public

visitNewExpression : IOmlNewExpression \xrightarrow{o} ()

visitNewExpression (pcmp) \triangleq
  ( dcl str : char\ast := " new ";
    printNodeField(pcmp.getName ());
    str := str \concat result \concat " ()";
    result := str
);

public
visitNumericLiteral : IOmlNumericLiteral \rightarrow ()
visitNumericLiteral (pcmp) \triangleq 
(  
dcl str : char* := "";
   printNatField(pcmp.getVal ());
   str := str \rightarrow result;
   result := str
);

public
visitOperationDefinitions : IOmlOperationDefinitions \rightarrow ()
visitOperationDefinitions (pcmp) \triangleq 
(  
dcl str : char* := nl \rightarrow "operations" \rightarrow nl \rightarrow nl;
   for db in pcmp.getOperationList ()
      do (  
         printNodeField(db);
            str := str \rightarrow result \rightarrow nl
      );
   if len pcmp.getOperationList () > 0
      then result := str
   else result := ""
);

public
visitOperationDefinition : IOmlOperationDefinition \rightarrow ()
visitOperationDefinition (pcmp) \triangleq 
(  
dcl str : char* := "";
   printNodeField(pcmp.getAccess ());
   str := str \rightarrow result;
   printNodeField(pcmp.getShape ());
   str := str \rightarrow result;
   result := str
);

public
visitExplicitOperation : IOmlExplicitOperation \rightarrow ()
visitExplicitOperation (pcmp) \triangleq 
(  
dcl str : char* := pcmp.getIdentifier () \rightarrow "" : "";
   printNodeField(pcmp.getType ());
   str := str \rightarrow result;
   str := str \rightarrow nl \rightarrow pcmp.getIdentifier () \rightarrow ""(";
   for db in pcmp.getParameterList ()
do (  
    printNodeField(db);  
    str := str \rightarrow result  
  );  
str := str \rightarrow " ";  
printNodeField(pcmp.getBody());  
str := str \rightarrow result \rightarrow " ";  
result := str
);  

public
visitOperationType : IOmlOperationType \rightarrow ()
visitOperationType (pcmp) \triangleq
  (  
    dcl str : char* := " ";  
    printNodeField(pcmp.getDomType());  
    str := str \rightarrow result \rightarrow " ";  
    printNodeField(pcmp.getRngType());  
    str := str \rightarrow result;  
    result := str
  );

public
visitOperationBody : IOmlOperationBody \rightarrow ()
visitOperationBody (pcmp) \triangleq
  (  
    dcl str : char* := " ";  
    if pcmp.hasStatement() then printNodeField(pcmp.getStatement())  
    else result := " ";  
    str := str \rightarrow result;  
    if pcmp.getNotYetSpecified() then str := str \rightarrow " is not yet specified";  
    if pcmp.getSubclassResponsibility() then str := str \rightarrow " sub class responsibility";  
    str := str \rightarrow " ";  
    result := str
  );

public
visitSkipStatement : IOmlSkipStatement \rightarrow ()
visitSkipStatement (_) \triangleq
  (  
    dcl str : char* := "skip";  
    result := str
  );

public
visitParameter : IOmlParameter $\xrightarrow{\alpha}$ ()
visitParameter (pcmp) $\xrightarrow{\alpha}$
  (  
    dcl str : char := "";
    for db in pcmp.getPatternList ()
      do (  
        printNodeField(db);
        str := str $\bowtie$ result $\bowtie$ "", "
      ) ;
    str := str (1,...,len str - 2);
    result := str
  ) ;

public

visitFunctionDefinitions : IOmlFunctionDefinitions $\xrightarrow{\alpha}$ ()
visitFunctionDefinitions (pcmp) $\xrightarrow{\alpha}$
  (  
    dcl str : char := nl $\bowtie$ "functions" $\bowtie$ nl $\bowtie$ nl;
    for db in pcmp.getFunctionList ()
      do (  
        printNodeField(db);
        str := str $\bowtie$ result $\bowtie$ nl
      ) ;
    result := str
  ) ;

public

visitFunctionDefinition : IOmlFunctionDefinition $\xrightarrow{\alpha}$ ()
visitFunctionDefinition (pcmp) $\xrightarrow{\alpha}$
  (  
    dcl str : char := "";
    printNodeField(pcmp.getAccess ()) ;
    str := str $\bowtie$ result;
    printNodeField(pcmp.getShape ()) ;
    str := str $\bowtie$ result;
    result := str
  ) ;

public

visitExplicitFunction : IOmlExplicitFunction $\xrightarrow{\alpha}$ ()
visitExplicitFunction (pcmp) $\xrightarrow{\alpha}$
  (  
    dcl str : char := pcmp.getIdentifier () $\bowtie$ "", ";
    for db in pcmp.getTypeVariableList ()
  ) ;
do ( printNodeField(db);  
    str := str ▼ result  
  ) ;
printNodeField(pcmp.getType());  
str := str ▼ result;
str := str ▼ nl ▼ pcmp.getIdentifier() ▼ "(";
for db in pcmp.getParameterList() do (  
    printNodeField(db);  
    str := str ▼ result  
  ) ;
str := str ▼ ") == is not yet specified;";
result := str  
);

public
visitPartialFunctionType : IOmlPartialFunctionType ▼ ()
visitPartialFunctionType (pcmp) △
(  
    dcl str : char* := "";
    printNodeField(pcmp.getDomType());  
    str := str ▼ result ▼ " - > ";
    printNodeField(pcmp.getRngType());  
    str := str ▼ result;
    result := str
  );

public
visitUnionType : IOmlUnionType ▼ ()
visitUnionType (pcmp) △
(  
    dcl str : char* := "";
    pcmp.getLhsType() .accept(self);  
    str := str ▼ result;
    pcmp.getRhsType() .accept(self);  
    str := str ▼ " | " ▼ result;
    result := str
  );

public
visitProductType : IOmlProductType ▼ ()
visitProductType (pcmp) △
(  
    dcl str : char* := "";

\begin{verbatim}
pcmp.getLhsType ( ) accept(self);
str := str \triangleright result;

pcmp.getRhsType ( ) accept(self);
str := str \triangleright " * " \triangleright result;
result := str;

public

visitTraceDefinitions : IOmlTraceDefinitions \to ()
visitTraceDefinitions (pcmp) \triangleq
(dcl str : char* := nl \triangleright "traces" \triangleright nl \triangleright nl;
for db in pcmp.getTraces ()
do ( printNodeField(db);
str := str \triangleright result \triangleright nl
);
result := str;

public

visitNamedTrace : IOmlNamedTrace \to ()
visitNamedTrace (pcmp) \triangleq
(dcl str : char* := " ";
str := str \triangleright pcmp.getName () \triangleright " : ";
printNodeField(pcmp.getDefs ());
str := str \triangleright result;
result := str;

public

visitTraceDefinition : IOmlTraceDefinition \to ()
visitTraceDefinition (pNode) \triangleq pNode.
accept(self);

public

visitTraceDefinitionItem : IOmlTraceDefinitionItem \to ()
visitTraceDefinitionItem (pcmp) \triangleq
(dcl str : char* := " ";
printSeqofField(pcmp.getBind ());
str := str \triangleright result;
printNodeField(pcmp.getTest ());
str := str \triangleright result;
if pcmp.hasRegexpr ()
then printNodeField(pcmp.getRegexpr ());
\end{verbatim}
else result := "";
    str := str ↟ result;
    result := str
);

public
visitTraceBinding : IOmlTraceBinding ↦ ()
visitTraceBinding (pNode) △ pNode.
    accept(self);

public
visitTraceLetBinding : IOmlTraceLetBinding ↦ ()
visitTraceLetBinding (pcmp) △
    ( dcl str : char* := "";
        printSeqofField(pcmp.getDefinitionList ());
        str := str ↟ result;
        result := str
    );

public
visitTraceBracketedDefinition : IOmlTraceBracketedDefinition ↦ ()
visitTraceBracketedDefinition (pcmp) △
    ( dcl str : char* := "(");
        printNodeField(pcmp.getDefinition ());
        str := str ↟ result ↟ ")";
        result := str
    );

public
visitTraceMethodApply : IOmlTraceMethodApply ↦ ()
visitTraceMethodApply (pcmp) △
    ( dcl str : char* := "";
        str := str ↟ pcmp.getVariableName () ↟ ",";
        str := str ↟ pcmp.getMethodName () ↟ "(";
        printSeqofField(pcmp.getArgs ());
        str := str ↟ result ↟ ")";
        result := str
    );

public
visitTraceCoreDefinition : IOmlTraceCoreDefinition ↦ ()
visitTraceCoreDefinition (pNode) △ pNode.
    accept(self);
visitTraceRepeatPattern : IOmlTraceRepeatPattern $\rightarrow$ ()
visitTraceRepeatPattern (pNode) $\triangleq$ pNode.
    accept(self);

public

visitTraceZeroOrMore : IOmlTraceZeroOrMore $\rightarrow$ ()
visitTraceZeroOrMore (-) $\triangleq$
    (    dcl str : char* := "*";
    result := str
    );

public

visitTraceOneOrMore : IOmlTraceOneOrMore $\rightarrow$ ()
visitTraceOneOrMore (-) $\triangleq$
    (    dcl str : char* := " + ";
    result := str
    );

public

visitTraceZeroOrOne : IOmlTraceZeroOrOne $\rightarrow$ ()
visitTraceZeroOrOne (-) $\triangleq$
    (    dcl str : char* := "?";
    result := str
    );

public

visitTraceRange : IOmlTraceRange $\rightarrow$ ()
visitTraceRange (pcmp) $\triangleq$
    (    dcl str : char* := "{ ";
    printNodeField(pcmp.getLower ());
    str := str $\rightarrow$ result;
    if pcmp.hasUpper ()
    then (        printNodeField(pcmp.getUpper ());
        str := str $\rightarrow$ " ", " $\rightarrow$ result
    )
    ;
    str := str $\rightarrow$ "}";
    result := str
    );

public

visitTraceChoiceDefinition : IOmlTraceChoiceDefinition $\rightarrow$ ()
visitTraceChoiceDefinition (pcmp) $\triangleq$
    (    dcl str : char* := " ",
    count : N := 1;
    for db in pcmp.getDefs ()
do (  printNodeField(db);  
  if len pcmp.getDefs() = count  
  then str := str \cup result 
  else str := str \cup result \cup " | ";  
  count := count + 1  
);  
result := str  
);  

public  
visitTraceSequenceDefinition : IOmlTraceSequenceDefinition \rightarrow ()  
visitTraceSequenceDefinition (pcmp) \triangleq  
(  dcl str : char* := "," ,  
count : N := 1;  
for db in pcmp.getDefs()  
do (  printNodeField(db);  
  if len pcmp.getDefs() = count  
  then str := str \cup result 
  else str := str \cup result \cup " | ";  
  count := count + 1  
);  
result := str  
)  
end Oml2VppVisitor  

Test Suite : vdm.tc  
Class : Oml2VppVisitor  

<table>
<thead>
<tr>
<th>Name</th>
<th>#Calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oml2VppVisitor'visitName</td>
<td>9</td>
<td>70%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitNode</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitType</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor'printField</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitClass</td>
<td>15</td>
<td>89%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitScope</td>
<td>23</td>
<td>93%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitIntType</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitLiteral</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitNatType</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitPattern</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitSetType</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor'printNatField</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitCharType</td>
<td>4</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor'visitDocument</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Name</td>
<td>#Calls</td>
<td>Coverage</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>Om12VppVisitor`visitNat1Type</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitRealType</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitSeq0Type</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitSeq1Type</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTypeName</td>
<td>9</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`printBoolField</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`printCharField</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`printNodeField</td>
<td>251</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`printRealField</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitEmptyType</td>
<td>10</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitParameter</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitUnionType</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`printSeqofField</td>
<td>14</td>
<td>35%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitExpression</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitNamedTrace</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitSimpleType</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTraceRange</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitValueShape</td>
<td>1</td>
<td>95%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitValueShape</td>
<td>1</td>
<td>95%</td>
</tr>
<tr>
<td>Om12VppVisitor`printStringField</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitProductType</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTextLiteral</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTraceBinding</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitNewExpression</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitOperationBody</td>
<td>5</td>
<td>73%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitOperationType</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitSkipStatement</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitNumericLiteral</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitSpecifications</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTraceOneOrMore</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTraceZeroOrOne</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTypeDefinition</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTraceDefinition</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTraceLetBinding</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTraceZeroOrMore</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitTypeDefinitions</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitValueDefinition</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>Om12VppVisitor`visitAccessDefinition</td>
<td>23</td>
<td>90%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitExplicitFunction</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Om12VppVisitor`visitInstanceVariable</td>
<td>17</td>
<td>√</td>
</tr>
<tr>
<td>Name</td>
<td>#Calls</td>
<td>Coverage</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceDefinitions</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceMethodApply</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitValueDefinitions</td>
<td>15</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitExplicitOperation</td>
<td>5</td>
<td>86%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitInheritanceClause</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitPatternIdentifier</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitFunctionDefinition</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceRepeatPattern</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitFunctionDefinitions</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitOperationDefinition</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitPartialFunctionType</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceCoreDefinition</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceDefinitionItem</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitAssignmentDefinition</td>
<td>17</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitOperationDefinitions</td>
<td>15</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceChoiceDefinition</td>
<td>3</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceSequenceDefinition</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitTraceBracketedDefinition</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitSymbolicLiteralExpression</td>
<td>5</td>
<td>√</td>
</tr>
<tr>
<td>Oml2VppVisitor`visitInstanceVariableDefinitions</td>
<td>19</td>
<td>√</td>
</tr>
<tr>
<td><strong>Total Coverage</strong></td>
<td></td>
<td><strong>63%</strong></td>
</tr>
</tbody>
</table>
In this appendix the OML AST is listed.

```plaintext
%prefix Oml;
%package org.overturetool.ast;
%directory "c:\COMU\buildOml";
%top Specifications Expression;

Specifications ::
  class_list : seq of Class;

Class ::
  identifier : Identifier
  generic_types : seq of Type
  inheritance_clause : [InheritanceClause]
  class_body : seq of DefinitionBlock
  system_spec : bool;

InheritanceClause ::
  identifier_list : seq of Identifier;

DefinitionBlock =
  TypeDefinitions |
  ValueDefinitions |
  FunctionDefinitions |
```
APPENDIX G. OML AST

OperationDefinitions | InstanceVariableDefinitions | SynchronizationDefinitions | ThreadDefinition | TraceDefinitions;

----
---- TYPE DEFINITIONS
----

TypeDefinitions ::
   type_list : seq of TypeDefinition;

TypeDefinition ::
   access : AccessDefinition
   shape : TypeShape;

AccessDefinition ::
   async_access : bool
   static_access : bool
   scope : Scope;

Scope =
   <PUBLIC> | <PRIVATE> | <PROTECTED> | <DEFAULT>;

TypeShape =
   SimpleType | ComplexType;

SimpleType ::
   identifier : Identifier
   type : Type
   invariant : [Invariant];

ComplexType ::
   identifier : Identifier
   field_list : seq of Field
   invariant : [Invariant];

Type =
   BracketedType |
   BoolType |
   NatType |
   Nat1Type |
   IntType |
   RatType |
   RealType |
   CharType |
Invariant ::
  pattern : Pattern
  expression : Expression;

BracketedType ::
  type : Type;

BoolType :: ;

NatType :: ;

Nat1Type :: ;

IntType :: ;

RatType :: ;

RealType :: ;

CharType :: ;

TokenType :: ;

QuoteType ::
  quote_literal : QuoteLiteral;

CompositeType ::
  identifier : Identifier

-- added for Thomas Christensen
field_list : seq of Field;

Field ::
  identifier : [seq of char]
  type : Type
  ignore : bool;

UnionType ::
  lhs_type : Type
  rhs_type : Type;

ProductType ::
  lhs_type : Type
  rhs_type : Type;

OptionalType ::
  type : Type;

SetType ::
  type : Type;

Seq0Type ::
  type : Type;

Seq1Type ::
  type : Type;

GeneralMapType ::
  dom_type : Type
  rng_type : Type;

InjectiveMapType ::
  dom_type : Type
  rng_type : Type;

PartialFunctionType ::
  dom_type : Type
  rng_type : Type;

TotalFunctionType ::
  dom_type : Type
  rng_type : Type;

OperationType ::
  dom_type : Type
  rng_type : Type;
EmptyType ::;

TypeName ::
   name : Name;

TypeVariable ::
   identifier : Identifier;

ClassTypeInstantiation ::
   name : Name
   generic_types : seq of Type;

---
--- VALUE DEFINITIONS
---

ValueDefinitions ::
   value_list : seq of ValueDefinition;

ValueDefinition ::
   access : AccessDefinition
   shape : ValueShape;

ValueShape ::
   pattern : Pattern
   type : [Type]
   expression : Expression;

---
--- FUNCTION DEFINITIONS
---

FunctionDefinitions ::
   function_list : seq of FunctionDefinition;

FunctionDefinition ::
   access : AccessDefinition
   shape : FunctionShape;

FunctionShape =
   ExplicitFunction |
   ImplicitFunction |
   ExtendedExplicitFunction |
   TypelessExplicitFunction; -- added for Thomas Christensen

ExplicitFunction ::
   identifier : Identifier
type_variable_list : seq of TypeVariable
type : Type
parameter_list : seq of Parameter
body : FunctionBody
trailer : FunctionTrailer;

Parameter ::
  pattern_list : seq of Pattern;

ImplicitFunction ::
  identifier : Identifier
type_variable_list : seq of TypeVariable
pattern_type_pair_list : seq of PatternTypePair
identifier_type_pair_list : seq of IdentifierTypePair
trailer : FunctionTrailer;

PatternTypePair ::
  pattern_list : seq of Pattern
type : Type;

IdentifierTypePair ::
  identifier : Identifier
type : Type;

ExtendedExplicitFunction ::
  identifier : Identifier
type_variable_list : seq of TypeVariable
pattern_type_pair_list : seq of PatternTypePair
identifier_type_pair_list : seq of IdentifierTypePair
body : FunctionBody
trailer : FunctionTrailer;

TypelessExplicitFunction ::
  identifier : Identifier
  parameter_list : seq of Parameter
  body : FunctionBody
  trailer : FunctionTrailer;

FunctionBody ::
  function_body : [Expression]
  not_yet_specified : bool
  subclass_responsibility : bool;

FunctionTrailer ::
  pre_expression : [Expression]
  post_expression : [Expression];
---
--- OPERATION DEFINITIONS
---

OperationDefinitions ::
    operation_list : seq of OperationDefinition;

OperationDefinition ::
    access : AccessDefinition
    shape : OperationShape;

OperationShape =
    ExplicitOperation | ImplicitOperation | ExtendedExplicitOperation;

ExplicitOperation ::
    identifier : Identifier
    type : Type
    parameter_list : seq of Pattern
    body : OperationBody
    trailer : OperationTrailer;

ImplicitOperation ::
    identifier : Identifier
    pattern_type_pair_list : seq of PatternTypePair
    identifier_type_pair_list : seq of IdentifierTypePair
    trailer : OperationTrailer;

ExtendedExplicitOperation ::
    identifier : Identifier
    pattern_type_pair_list : seq of PatternTypePair
    identifier_type_pair_list : seq of IdentifierTypePair
    body : OperationBody
    trailer : OperationTrailer;

OperationBody ::
    statement : [Statement]
    not_yet_specified : bool
    subclass_responsibility : bool;

OperationTrailer ::
    externals : [Externals]
    pre_expression : [Expression]
    post_expression : [Expression]
    exceptions : [Exceptions];
Externals ::
   ext_list : seq of VarInformation;

VarInformation ::
   mode : Mode
   name_list : seq of Name
   type : [Type];

Mode =
   <RD> \ Or <WR>;

Exceptions ::
   error_list : seq of Error;

Error ::
   identifier : Identifier
   lhs : Expression
   rhs : Expression;

---
--- INSTANCE VARIABLES
---

InstanceVariableDefinitions ::
   variables_list : seq of InstanceVariableShape;

InstanceVariableShape =
   InstanceVariable |
   InstanceVariableInvariant;

InstanceVariable ::
   access : AccessDefinition
   assignment_definition : AssignmentDefinition;

InstanceVariableInvariant ::
   invariant : Expression;

---
--- SYNCHRONIZATION
---

SynchronizationDefinitions ::
   sync_list : seq of SyncPredicate;

SyncPredicate =
   PermissionPredicate |
   MutexPredicate |
MutexAllPredicate ;

PermissionPredicate ::
  name : Name
  expression : Expression;

MutexPredicate ::
  name_list : seq of Name;

MutexAllPredicate ::;

---
--- THREAD DEFINITIONS
---

ThreadDefinition ::
  thread_specification : [ThreadSpecification];

ThreadSpecification =
  PeriodicThread |
  SporadicThread |
  ProcedureThread ;

PeriodicThread ::
  args : seq of Expression
  name : Name;

SporadicThread :: -- added for Marcel Verhoef
  args : seq of Expression
  name : Name;

ProcedureThread ::
  statement : Statement;

---
--- TRACE DEFINITIONS (added for Adriana Sucena)
---

TraceDefinitions ::
  traces : seq of NamedTrace;

NamedTrace ::
  name : seq of char
  defs : TraceDefinition;

TraceDefinition =
  TraceDefinitionItem |
TraceSequenceDefinition | TraceChoiceDefinition;

TraceSequenceDefinition ::
defs : seq of TraceDefinition;

TraceChoiceDefinition ::
defs : seq of TraceDefinition;

TraceDefinitionItem ::
bind : seq of TraceBinding
test : TraceCoreDefinition
reexpr : [TraceRepeatPattern];

TraceBinding =
TraceLetBinding | TraceLetBeBinding;

TraceLetBinding ::
definition_list : seq of ValueShape;

TraceLetBeBinding ::
bind : Bind
best : [Expression];

TraceCoreDefinition =
TraceMethodApply | TraceBracketedDefinition;

TraceMethodApply ::
variable_name : Identifier
method_name : Identifier
args : seq of Expression;

TraceBracketedDefinition ::
definition : TraceDefinition;

TraceRepeatPattern =
TraceZeroOrMore | TraceOneOrMore | TraceZeroOrOne | TraceRange;

TraceZeroOrMore :: ;

TraceOneOrMore :: ;
TraceZeroOrOne :: ;

TraceRange ::
  lower : NumericLiteral
  upper : [NumericLiteral];

---

--- EXPRESSIONS
---

Expression =
  BracketedExpression |
  LetExpression |
  LetBeExpression |
  DefExpression |
  IfExpression |
  CasesExpression |
  UnaryExpression |
  BinaryExpression |
  ForAllExpression |
  ExistsExpression |
  ExistsUniqueExpression |
  IotaExpression |
  TokenExpression |
  SetEnumeration |
  SetComprehension |
  SetRangeExpression |
  SequenceEnumeration |
  SequenceComprehension |
  SubsequenceExpression |
  MapEnumeration |
  MapComprehension |
  TupleConstructor |
  RecordConstructor |
  MuExpression |
  ApplyExpression |
  FieldSelect |
  FunctionTypeSelect |
  FunctionTypeInstantiation |
  LambdaExpression |
  NewExpression |
  SelfExpression |
  ThreadIdExpression |
  TimeExpression |
  IsExpression |
  UndefinedExpression |
  PreconditionExpression |
IsofbaseclassExpression | IsofclassExpression | SamebaseclassExpression | SameclassExpression | ReqExpression | ActExpression | FinExpression | ActiveExpression | WaitingExpression | Name | OldName | SymbolicLiteralExpression;

BracketedExpression ::
  expression : Expression;

LetExpression ::
  definition_list : seq of ValueShape
  expression : Expression;

LetBeExpression ::
  bind : Bind
  best : [Expression]
  expression : Expression;

DefExpression ::
  pattern_bind_list : seq of PatternBindExpression
  expression : Expression;

PatternBindExpression ::
  pattern_bind : PatternBind
  expression : Expression;

IfExpression ::
  if_expression : Expression
  then_expression : Expression
  elseif_expression_list : seq of ElseIfExpression
  else_expression : Expression;

ElseIfExpression ::
  elseif_expression : Expression
  then_expression : Expression;

CasesExpression ::
  match_expression : Expression
  alternative_list : seq of CasesExpressionAlternative
  others_expression : [Expression];
CasesExpressionAlternative ::
    pattern_list : seq of Pattern
    expression : Expression;

UnaryExpression ::
    operator : UnaryOperator
    expression : Expression;

UnaryOperator =
    <PLUS> |
    <MINUS> |
    <ABS> |
    <FLOOR> |
    <NOT> |
    <CARD> |
    <POWER> |
    <DUNION> |
    <DINTER> |
    <HD> |
    <TL> |
    <LEN> |
    <ELEMS> |
    <INDS> |
    <DCONC> |
    <DOM> |
    <RNG> |
    <DMERGE> |
    <INVERSE>;

BinaryExpression ::
    lhs_expression : Expression
    operator : BinaryOperator
    rhs_expression : Expression;

BinaryOperator =
    <PLUS> |
    <MINUS> |
    <MULTIPLY> |
    <DIVIDE> |
    <DIV> |
    <REM> |
    <MOD> |
    <LT> |
    <LE> |
    <GT> |
    <GE> |
<EQ> |  
<NE> |  
<OR> |  
<AND> |  
<IMPLY> |  
<EQUIV> |  
<INSET> |  
<NOTINSET> |  
<SUBSET> |  
<PSUBSET> |  
<UNION> |  
<Ddifference> |  
<INTER> |  
<CONC> |  
<MODIFY> |  
<Munion> |  
<mapDOMRESTO> |  
<mapDOMRESBY> |  
<MApRNGRESTO> |  
<MApRNGRESBY> |  
<COMP> |  
<ITERATE> |  
<TUPSEL> ;

ForAllExpression ::
   bind_list : seq of Bind
   expression : Expression;

ExistsExpression ::
   bind_list : seq of Bind
   expression : Expression;

ExistsUniqueExpression ::
   bind : Bind
   expression : Expression;

IotaExpression ::
   bind : Bind
   expression : Expression;

TokenExpression ::
   expression : Expression;

SetEnumeration ::
   expression_list : seq of Expression;

SetComprehension ::
expression : Expression
bind_list : seq of Bind
guard : [Expression];

SetRangeExpression ::
  lower : Expression
  upper : Expression;

SequenceEnumeration ::
  expression_list : seq of Expression;

SequenceComprehension ::
  expression : Expression
  set_bind : SetBind
  guard : [Expression];

SubsequenceExpression ::
  expression : Expression
  lower : Expression
  upper : Expression;

MapEnumeration ::
  maplet_list : seq of Maplet;

Maplet ::
  dom_expression : Expression
  rng_expression : Expression;

MapComprehension ::
  expression : Maplet
  bind_list : seq of Bind
  guard : [Expression];

TupleConstructor ::
  expression_list : seq of Expression;

RecordConstructor ::
  name : Name
  expression_list : seq of Expression;

MuExpression ::
  expression : Expression
  modifier_list : seq of RecordModifier;

RecordModifier ::
  identifier : Identifier
  expression : Expression;
ApplyExpression ::
  expression : Expression
  expression_list : seq of Expression;

FieldSelect ::
  expression : Expression
  name : Name;

FunctionTypeSelect ::
  expression : Expression
  function_type_instantiation : FunctionTypeInstantiation;

FunctionTypeInstantiation ::
  name : Name
  type_list : seq of Type;

LambdaExpression ::
  type_bind_list : seq of TypeBind
  expression : Expression;

NewExpression ::
  name : Name
  generic_types : seq of Type
  -- added for Thomas Christensen
  expression_list : seq of Expression;

SelfExpression :: ;

ThreadIdExpression :: ;

TimeExpression :: ;

IsExpression ::
  type : Type
  expression : Expression;

UndefinedExpression :: ;

PreconditionExpression ::
  expression_list : seq of Expression;

IsofbaseclassExpression ::
  name : Name
  expression : Expression;

IsofclassExpression ::
  name : Name
expression : Expression;

SamebaseclassExpression ::
    lhs_expression : Expression
    rhs_expression : Expression;

SameclassExpression ::
    lhs_expression : Expression
    rhs_expression : Expression;

ReqExpression ::
    name_list : seq of Name;

ActExpression ::
    name_list : seq of Name;

FinExpression ::
    name_list : seq of Name;

ActiveExpression ::
    name_list : seq of Name;

WaitingExpression ::
    name_list : seq of Name;

Name ::
    class_identifier : [Identifier]
    identifier : Identifier;

OldName ::
    identifier : Identifier;

SymbolicLiteralExpression ::
    literal : Literal;

---
--- STATEMENTS
---

Statement =
    LetStatement |
    LetBeStatement |
    DefStatement |
    BlockStatement |
    DclStatement |
    AssignStatement |
    AtomicStatement |
IfStatement | CasesStatement | SequenceForLoop | SetForLoop | IndexForLoop | WhileLoop | NondeterministicStatement | CallStatement | ReturnStatement | SpecificationStatement | StartStatement | DurationStatement | CyclesStatement | AlwaysStatement | TrapStatement | RecursiveTrapStatement | ExitStatement | ErrorStatement | SkipStatement ;

LetStatement ::
    definition_list : seq of ValueShape
    statement : Statement;

LetBeStatement ::
    bind : Bind
    best : [Expression]
    statement : Statement;

DefStatement ::
    definition_list : seq of EqualsDefinition
    statement : Statement;

EqualsDefinition ::
    pattern_bind : PatternBind
    expression : Expression;

BlockStatement ::
    dcl_statement_list : seq of DclStatement
    statement_list : seq of Statement;

DclStatement ::
    definition_list : seq of AssignmentDefinition;

AssignmentDefinition ::
    identifier : Identifier
    type : Type
expression : [Expression];

AssignStatement ::
  state_designator : StateDesignator
  expression : Expression;

AtomicStatement ::
  assignment_list : seq of AssignStatement;

StateDesignator =
  StateDesignatorName | FieldReference | MapOrSequenceReference;

StateDesignatorName ::
  name : Name;

FieldReference ::
  state_designator : StateDesignator
  identifier : Identifier;

MapOrSequenceReference ::
  state_designator : StateDesignator
  expression : Expression;

IfStatement ::
  expression : Expression
  then_statement : Statement
  elseif_statement : seq of ElseIfStatement
  else_statement : [Statement];

ElseIfStatement ::
  expression : Expression
  statement : Statement;

CasesStatement ::
  match_expression : Expression
  alternative_list : seq of CasesStatementAlternative
  others_statement : [Statement];

CasesStatementAlternative ::
  pattern_list : seq of Pattern
  statement : Statement;

SequenceForLoop ::
  pattern_bind : PatternBind
  in_reverse : bool
expression : Expression
statement : Statement;

SetForLoop ::
  pattern : Pattern
  expression : Expression
  statement : Statement;

IndexForLoop ::
  identifier : Identifier
  init_expression : Expression
  limit_expression : Expression
  by_expression : [Expression]
  statement : Statement;

WhileLoop ::
  expression : Expression
  statement : Statement;

NondeterministicStatement ::
  statement_list : seq of Statement;

CallStatement ::
  object_designator : [ObjectDesignator]
  name : Name
  expression_list : seq of Expression;

ObjectDesignator =
  ObjectDesignatorExpression |
  ObjectFieldReference |
  ObjectApply;

ObjectDesignatorExpression ::
  expression : Expression;

ObjectFieldReference ::
  object_designator : ObjectDesignator
  name : Name;

ObjectApply ::
  object_designator : ObjectDesignator
  expression_list : seq of Expression;

ReturnStatement ::
  expression : [Expression];

SpecificationStatement ::
externals : [Externals]
pre_expression : [Expression]
post_expression : Expression
exceptions : [Exceptions];

StartStatement ::
  expression : Expression;

DurationStatement ::
  duration_expression : seq of Expression
  statement : Statement;

CyclesStatement ::
  cycles_expression : seq of Expression
  statement : Statement;

AlwaysStatement ::
  always_part : Statement
  in_part : Statement;

TrapStatement ::
  pattern_bind : PatternBind
  with_part : Statement
  in_part : Statement;

RecursiveTrapStatement ::
  trap_list : seq of TrapDefinition
  in_part : Statement;

TrapDefinition ::
  pattern_bind : PatternBind
  statement : Statement;

ExitStatement ::
  expression : [Expression];

ErrorStatement ::;

SkipStatement ::;

---
--- PATTERNS
---

Pattern =
  DontCarePattern |
  PatternIdentifier |
DontCarePattern ::;

PatternIdentifier ::
  identifier : Identifier;

MatchValue ::
  expression : Expression;

SymbolicLiteralPattern ::
  literal : Literal;

SetEnumPattern ::
  pattern_list : seq of Pattern;

SetUnionPattern ::
  lhs_pattern : Pattern
  rhs_pattern : Pattern;

SeqEnumPattern ::
  pattern_list : seq of Pattern;

SeqConcPattern ::
  lhs_pattern : Pattern
  rhs_pattern : Pattern;

TuplePattern ::
  pattern_list : seq of Pattern;

RecordPattern ::
  name : Name
  pattern_list : seq of Pattern;

---

BINDINGS
---

PatternBind =
  Pattern |
Bind;

Bind =
  SetBind |
  TypeBind;

-- SetBind is used for both single and multiple binds

SetBind :::
  pattern : seq of Pattern
  expression : Expression;

-- TypeBind is used for both single and multiple binds

TypeBind :::
  pattern : seq of Pattern
  type : Type;

---
--- LEXICAL ELEMENTS
---

Literal =
  NumericLiteral |
  RealLiteral |
  BooleanLiteral |
  NilLiteral |
  CharacterLiteral |
  TextLiteral |
  QuoteLiteral;

NumericLiteral :::
  val : nat;

RealLiteral :::
  val : real;

BooleanLiteral :::
  val : bool;

NilLiteral ::;

CharacterLiteral :::
  val : char;

TextLiteral :::
  val : seq of char;
QuoteLiteral ::
    val : seq of char;

Identifier = seq of char

Listing G.1: OML AST
Appendix H

UML AST

In this appendix the UML AST is listed.

```
%prefix Uml;
%package org.overturetool.uml.ast;
%directory "C:\COMU\build";
%top Model;

Model ::
  name : String
  definitions : set of ModelElement;

ModelElement = Class | Association |
                  Constraint | Collaboration;

Class ::
  name : String
  classBody : set of DefinitionBlock
  isAbstract : bool
  superClass : seq of ClassNameType
```
visibility : VisibilityKind
isStatic : bool
isActive : bool
templatesignature : [TemplateSignature];

VisibilityKind = <PUBLIC> | <PRIVATE> | <PROTECTED> ;

TemplateSignature :::
  templateParameters : set of TemplateParameter;

TemplateParameter :::
  name : String;

DefinitionBlock =
  OwnedOperations | OwnedProperties | NestedClassifiers;

----------------------------------------
-- Operation
----------------------------------------

OwnedOperations :::
  operationList : set of Operation;

Operation :::
  name : String
  visibility : VisibilityKind
  multiplicity : MultiplicityElement --aka return type
  isQuery : bool
  type : [Type] --aka return type
  isStatic : bool
  ownedParameters : [Parameters];

Parameters :::
  parameterList : seq of Parameter;

Parameter :::
  name : String
  type : Type
  multiplicity : MultiplicityElement
  default : String
  direction : ParameterDirectionKind;

ParameterDirectionKind = <IN> | <INOUT> | <OUT> | <RETURN>;

MultiplicityElement :::
  isOrdered : bool
  isUnique : bool
  lower : nat
upper : [nat];

-- Property

OwnedProperties ::
  propetyList : set of Property;

Property ::
  name : String
  visibility : VisibilityKind
  multiplicity : [MultiplicityElement]
  type : Type
  isReadOnly : [bool]
  default : [ValueSpecification]
  isComposite : bool
  isDerived : [bool]
  isStatic : [bool]
  ownerClass : String
  qualifier : [Type];

-- Types

NestedClassifiers ::
  typeList : set of Type;

Type =
  BoolType |
  IntegerType |
  StringType |
  UnlimitedNatural |
  VoidType |
  CharType |
  ClassNameType;

BoolType ::;
IntegerType ::;
StringType ::;
UnlimitedNatural ::;
VoidType :: ;
CharType :: ;

ClassNameType ::
  name : String;

------------------------


-- Association
-----------------------------
Association ::
    ownedEnds : set of Property
    ownedNavigableEnds : set of Property
    name : [String]
    id : Id;
-----------------------------

-- Constraint
-----------------------------
Constraint ::
    constraintElements : set of Id
    specification : ValueSpecification;

ValueSpecification = LiteralString | LiteralInteger;
-----------------------------

-- Diagrams
-----------------------------
Collaboration :: -- Unknown in superstructure
    ownedBehavior : set of Interaction;

Interaction ::
    name : String
    lifelines : set of LifeLine
    fragments : set of InteractionFragment
    messages : seq of Message;

LifeLine ::
    name : String
    represents : [Type]; -- ConnectableElement - TypedElement - Class
    -- coveredBy : set of InteractionFragment;

InteractionFragment = OccurrenceSpecification | InteractionOperand | CombinedFragment | ExecutionSpecification;

OccurrenceSpecification = Mos; -- Mos = MessageOccurrenceSpecification
ExecutionSpecification = Bes; -- Bes = BehaviorExecutionSpecification

Mos ::
    name : String
    message : [Message]
covered : LifeLine

event : [CallEvent];

CallEvent ::
  operation : Operation;

Bes ::
  name : String
  startOs : OccurrenceSpecification
  finishOs : OccurrenceSpecification
  covered : set of LifeLine;

CombinedFragment ::
  name : String
  interactionOperator : InteractionOperatorKind
  operand : seq of InteractionOperand --seq1
  covered : set of LifeLine;

InteractionOperatorKind = <ALT> | <LOOP>;

InteractionOperand ::
  name : String
  fragments: seq of InteractionFragment
  covered : set of Mos --LifeLine
  guard : [InteractionConstraint];
  --inv io == forall cf in set elems io.fragments & is_CombinedFragment(cf)
  --    and forall mos in set io.covered
  --    & forall childMos in set dunion {op.covered | op in set elems cf.operand}
  --    & mos <> childMos;

InteractionConstraint ::
  minint : [ValueSpecification]
  maxint : [ValueSpecification];

Message ::
  name : String
  messageKind : MessageKind
  messageSort : MessageSort
  sendEvent : Mos
  sendReceive : Mos
  argument : seq of ValueSpecification;

MessageKind = <COMPLETE> | <UNKNOWN>;
MessageSort = <SYNCHCALL> | <ASYNCHCALL>;

LiteralString ::
  value : String;
LiteralInteger ::
   value: nat;

-----------------------------
-- Others
-----------------------------
String = seq of char;
Id = String

Listing H.1: UML AST
Appendix I

Features supported by Transformation

In this appendix a table I.1 shows a overview of the supported features by the VDM - UML transformation. An X denotes that the feature is fully specified. An (X) denotes that the feature is partly specified.
<table>
<thead>
<tr>
<th>Name (VDM)</th>
<th>Rule #</th>
<th>AST</th>
<th>VDM→UML</th>
<th>UML→VDM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td>14</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Functions</td>
<td>17</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td>17</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Generic classes</td>
<td>16</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>6, 5</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Instance variables</td>
<td>6, 5</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Initial value</td>
<td>7</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Visibility</td>
<td>2</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Thread</td>
<td>13</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract Class</td>
<td>15</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Access</td>
<td>3</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Types</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product Types</td>
<td>10</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Union Types</td>
<td>9</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Record Types</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optional Types</td>
<td>5</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Object Reference Types</td>
<td>5</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Collections and Relationships</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>map</td>
<td>12</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>11</td>
<td>X</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>seq</td>
<td>11</td>
<td>X</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>seq1</td>
<td>11</td>
<td>X</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>Traces</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core Definition</td>
<td>1819, 20, 24</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Definition List</td>
<td>21</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Choice Definition</td>
<td>22</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Repeat Pattern</td>
<td>23</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Bindings</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table I.1: Overview of supported features. X equals full support where (X) implies a partly support.
## List of Symbols and Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td>page 82</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree, a tree representation of the syntax of some source code</td>
<td>page 67</td>
</tr>
<tr>
<td>ASTGen</td>
<td>Converts an AST to VDM classes and Java interfaces. Maintained by the Overture project.</td>
<td>page 67</td>
</tr>
<tr>
<td>BES</td>
<td>Behavior Execution Specification, denoted the rectangle defining the execution time of a message in a SD</td>
<td>page 157</td>
</tr>
<tr>
<td>CD</td>
<td>UML Class Diagram</td>
<td>page 24</td>
</tr>
<tr>
<td>CSK</td>
<td>CSK is a Japanese conglomerate, owned by CSK Holdings Corporation</td>
<td>page 44</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph, a directed graph with no directed cycles</td>
<td>page 135</td>
</tr>
<tr>
<td>FM</td>
<td>Formal Method</td>
<td>page 9</td>
</tr>
<tr>
<td>GAO</td>
<td>Gesellschaft für Organisation</td>
<td>page 44</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
<td>page 43</td>
</tr>
<tr>
<td>IFAD</td>
<td>IFAD develops and markets simulation &amp; training products, networked simulation solutions, web and information technology solutions for civilian, military and Homeland Defence applications</td>
<td>page 44</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organization</td>
<td>page 43</td>
</tr>
<tr>
<td>JAR</td>
<td>Java ARChive, Java archive for source code, a optional manifest can specify which class to execute within the JAR file</td>
<td>page 111</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-driven architecture, a software design approach</td>
<td>page 15</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility, a meta-model used to formally define Unified Modeling Language (UML)</td>
<td>page 21</td>
</tr>
<tr>
<td>MOS</td>
<td>Message Occurrence Specification, links a message to other elements in a SD such as life lines and fragments</td>
<td>page 157</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group, the consortium responsible for CORBA architecture, Unified Modeling Language, and Model-driven architecture</td>
<td>page 19</td>
</tr>
<tr>
<td>OML</td>
<td>Overture Modeling Language, a specification language inspired by the object-oriented formal specification language VDM++ (Vienna Development Method)</td>
<td>page 17</td>
</tr>
<tr>
<td>OO</td>
<td>Object-oriented, a computer programming paradigm</td>
<td>page 19</td>
</tr>
<tr>
<td>RVL</td>
<td>Rose-VDM++ Link, integrates UML and VDM++ by providing a tight coupling between the VDM Toolbox and Rational Rose</td>
<td>page 31</td>
</tr>
<tr>
<td>SD</td>
<td>UML Sequence Diagram</td>
<td>page 23</td>
</tr>
<tr>
<td>SIC</td>
<td>Sensor Integration Controller</td>
<td>page 44</td>
</tr>
<tr>
<td>UIS</td>
<td>UML Infrastructure Specification</td>
<td>page 71</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
<td>page 19</td>
</tr>
<tr>
<td>USS</td>
<td>UML Superstructure Specification</td>
<td>page 71</td>
</tr>
<tr>
<td>VDM</td>
<td>Vienna Development Method</td>
<td>page 43</td>
</tr>
<tr>
<td>VDMEditor</td>
<td>Eclipse based editor for VDM</td>
<td>page 119</td>
</tr>
<tr>
<td>VDMJ</td>
<td>VDM tool implemented in Java by Nick Battle from Fujitsu</td>
<td>page 46</td>
</tr>
<tr>
<td>VDMTools</td>
<td>A development tool supporting precise modeling in the notations VDM-SL or VDM++</td>
<td>page 44</td>
</tr>
<tr>
<td>VDMUnit</td>
<td>VDM unit test framework</td>
<td>page 111</td>
</tr>
<tr>
<td>VICE</td>
<td>VDM In Constrained Environment, a recent research extension of</td>
<td>page 43</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange (XMI), is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML)</td>
<td>page 82</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language (XML)</td>
<td>page 40</td>
</tr>
</tbody>
</table>